



Politechnika Gdańska
**WYDZIAŁ ELEKTRONIKI,
TELEKOMUNIKACJI I INFORMATYKI**



Katedra: Katedra Architektury Systemów Komputerowych

Imię i nazwisko dyplomanta: **Piotr Kunowski**

Nr albumu: 106345

Forma i poziom studiów: Stacjonarne jednolite studia magisterskie

Kierunek studiów: Informatyka,
Aplikacje rozproszone i systemy internetowe

Praca dyplomowa magisterska

Temat pracy: **Wizualizacja ontologii zapisanych w języku OWL**

Kierujący pracą: dr inż. Piotr Szpryngier
Konsultant pracy: mgr inż. Tomasz Boiński

Zakres pracy:

W niniejszej pracy dyplomowej skupiono się na stworzeniu rozwiązania pozwalającego wizualizować ontologie.

W części teoretycznej pracy przedstawiono języki i sposoby zapisu ontologii oraz wyjaśniono zagadnienia związane z sieciami semantycznymi.

W części praktycznej pracy porównane zostały istniejące rozwiązania pozwalające na wizualizację ontologii. W oparciu o specyfikację wymagań stworzono projekt i zaimplementowano własną bibliotekę wizualizującą ontologie. Biblioteka zastała wdrożona do edytora OCS oraz Protégé i w tych rozwiązaniach została przetestowana.

Gdańsk, 2010

Spis treści

1. Wstęp	4
1.1. Motywacje	4
1.1.1. Zakres prac	5
1.2. Ontologie	5
1.3. Zastosowania ontologii	7
1.4. Reprezentacja wiedzy	8
2. Przegląd dostępnych rozwiązań do wizualizacji ontologii	18
2.1. Wstęp	18
2.2. Sposoby wizualizacji ontologii	18
2.3. Elementy wizualizacji	19
2.4. Przedstawienie wybranych rozwiązań do wizualizacji ontologii	20
2.5. Opis metody porównania	22
2.6. Test rozwiązań do wizualizacji ontologii	28
3. Analiza i projekt systemu	30
3.1. Wstęp	30
3.2. Cele systemu	30
3.3. Użytkownicy systemu	31
3.4. Wymagane funkcje systemu	31
3.5. Nazwa i logo tworzonej biblioteki	32
3.6. Wybór technologii	33
3.6.1. Wybór biblioteki graficznej	33
3.6.2. Format danych wejściowych	35
3.7. Planowanie projektu	35
3.7.1. Harmonogram projektu	35
3.7.2. Analiza SWOT	36
3.8. Projekt Wizualizacji	36
3.9. Analiza obiektowa	39
3.9.1. Diagram klas i pakietów	39
3.9.2. Zalecenie tworzenia biblioteki	40

4. Elementy implementacji	43
4.1. Wstęp	43
4.2. Biblioteka OWL API	43
4.3. Biblioteka Prefuse	44
4.4. Integracja z OWL API i Prefuse	46
4.5. Konwersja obiektu OWLOntology na struktury danych biblioteki Prefuse	48
4.5.1. Pełna wizualizacja	48
4.5.2. Wywnioskowana hierarchia klas i bytów	50
4.6. Plik właściwości	52
4.7. Plugin wizualizujący ontologie do Protégé	52
5. Testy aplikacji	54
5.1. Wstęp	54
5.2. Testy funkcjonalne	54
5.2.1. Testy funkcjonalne API dla programisty	55
5.2.2. Testy funkcjonalne związane z wizualizacją	56
5.3. Testy wydajnościowe	56
5.4. Testy wiarygodności	58
5.5. Testy użyteczności	59
5.6. Podsumowanie testów	60
6. Podsumowanie	61
Bibliografia	64

Rozdział 1

Wstęp

1.1. Motywacje

Szybki rozwój techniki komputerowej spowodował, że dyski komputerów na całym świecie przechowują ogromną liczbę danych. Internet jest wszechstronną składnicą informacji. Obecnie nie mamy problemu z brakiem wiedzy, ale z jej efektywnym wyszukiwaniem. Wiedza w Internecie jest w znacznym stopniu ukierunkowana na człowieka. Składowane dokumenty często nie posiadają kontekstu, są zapisane w postaci zrozumiałej tylko dla nas, np. graficznie. Komputery nie potrafią przetwarzać tych dokumentów. Konieczne jest więc tworzenie autonomicznych systemów klasyfikacji pojęć. Podkreślany jest też fakt, iż najlepszym sposobem przechowywania wiedzy są systemy zapisujące wiedzę w sposób zrozumiały zarówno dla człowieka, jak i dla komputera.

Jednym z rozwiązań, spełniających powyższe założenie, jest zaproponowana przez Bernersa Lee sieć semantyczna, w której dane powiązane są ze sobą znaczeniowo. Sieci semantyczne definiowane są jako rozszerzenie istniejących sieci, posiadających dobrze zdefiniowane informacje. Właściwa informacja została rozszerzona o znacznik semantyczny (kontekst), który maszyny mogą w łatwy sposób przetwarzać. Ponieważ znacznik semantyczny ma być rozumiany przez maszyny, musi posiadać formalnie zdefiniowany zapis. W3Cache Consortium zaproponowało ontologię, czyli w wielkim uproszczeniu reprezentację wiedzy poprzez zdefiniowanie zbioru pojęć i relacji pomiędzy nimi.

Ontologie bardzo szybko zyskały uznanie wśród inżynierów i badaczy z zakresu bezpieczeństwa i medycyny. Jednak okazało się, że ich budowa nie jest łatwym zadaniem. Nawet dla niewielkich rozwiązań liczba elementów ontologii jest bardzo duża. Na rynku wraz z zainteresowaniem ontologiami pojawiły się narzędzia służące

do ich tworzenia, edycji i zapisu. Autorzy edytorów ontologii, chcąc ułatwić pracę nad ontologiami, nieustannie ulepszają swoje rozwiązania. Aby móc sprawnie konstruować poprawne ontologie, konieczna jest ich wizualizacja, która znacznie ułatwia percepcję całego rozwiązania oraz umożliwia jego prawidłową ocenę.

1.1.1. Zakres prac

Celem tej pracy było stworzenie aplikacji pozwalającej na wizualizację ontologii zapisanych w języku OWL. W ramach pracy zostały wykonane:

1. **Projekt wizualizacji** - każdemu elementowi zdefiniowanemu w rekomendacji OWL DL został przypisany symbol graficzny.
2. **Projekt systemu** - została zaproponowana architektura aplikacji oraz dokonano wyboru pomocniczych bibliotek.
3. **Integracja z OCS** - biblioteka została wdrożona do systemu OCS.
4. **Plugin do Protégé** - po zapoznaniu się z API aplikacji Protégé, powstał plugin wizualizujący.

Efektem pracy są:

1. **Biblioteka SOVA**, która pozwala na wizualizację dostarczonych obiektów OWL API. Może zostać wykorzystana w dowolnym rozwiązaniu informatycznym. Posiada dobry i intuicyjny interfejs oraz dokumentację w javadocu.
 2. **Moduł wizualizacji OCS** jest rozszerzeniem edytora OCS o nowe możliwości wizualizacji, które dostarcza biblioteka SOVA.
 3. **Plugin do Protégé** pozwolił na pokazanie możliwości biblioteki SOVA w zewnętrznym rozwiązaniu oraz dostarczenie nowych opcji wizualizacji dla Protégé.
- Powyższe rozwiązania będą przydatne dla inżynierów ontologów, ułatwią im pracę nad ontologiami oraz pozwolą na łatwiejsze wykrywanie błędów.

1.2. Ontologie

Definicja ontologii

Jako pierwszy definicję ontologii w 1993 roku podał T. Gruber [9]. Określił on ontologię jako *specyfikację konceptualizacji*, czyli reprezentację pewnego wycinka wiedzy leżącego w zasięgu zainteresowania inżynierów. Zastosowanie ontologii w różnych

d dziedzinach i różnych społecznościach spowodowało powstanie wielu jej definicji. W sieciach semantycznych przyjęto poniższą definicję [8]:

ontologia *jest formalną, jawną specyfikacją wspólną konceptualizacji*

Aby w pełni zrozumieć definicję, wymagane jest wyjaśnienie użytych w niej słów:

- **formalna**

Ontologia jest wyrażona w języku reprezentacji wiedzy posiadającym formalną semantykę. Gwarantuje to, że ontologia może być przetwarzana przez maszyny i jednoznacznie interpretowana.

- **jawna**

Rodzaje użytych konceptów i ograniczenia i ich używanie są jawnie wyspecyfikowane. Niewyspecyfikowane elementy, pomimo iż są zrozumiałe dla człowieka, mogą nie być zrozumiałe dla przetwarzającej je maszyny.

- **wspólna**

Ontologia obejmuje wiedzę ogólnie uznaną, nie wiedzę prywatną jednostki. Aby wiedza zawarta w ontologii nie była subiektywnym spojrzeniem na jakieś zjawisko, często przy budowie ontologii bierze udział kilka osób.

- **konceptualizacja**

Ontologie określają wiedzę w sposób koncepcyjny, poprzez symbole reprezentujące koncepcje i relacje między nimi. Ponadto konceptualizacja ontologii pozwana na opis danej wiedzy w sposób ogólny, a nie tylko szczegółowy na poziomie konkretnych bytów.

Ontologie możemy klasyfikować wedle dwóch wymiarów [8]: poziomu szczegółowości lub poziomu zależności od zadania lub punktu widzenia. W pierwszym przypadku możemy mieć szczegółowo opisane ontologie oraz ontologie skierowane na możliwości wnioskowania, które charakteryzują się mniejszą szczegółowością. Drugi wymiar klasyfikacji pozwala wyróżnić: ontologię wysokiego poziomu, ontologię określającą dziedzinę, ontologię określającą zadanie oraz ontologię zastosowaną.

- *Ontologia wysokiego poziomu* określa podstawową, ogólną, niezależną od dziedziny wiedzę, np. opisują przestrzeń, czas, materię, wydarzenie itp.
- *Ontologia określająca dziedzinę lub zadanie* jest ontologią opisującą daną dziedzinę (np.: samochody, leki) lub zadania i działalności (np.: diagnozowanie, sprzedaż). Rozszerza ona pojęcia zdefiniowane w ontologii wysokiego poziomu o ich specyfikę w danej dziedzinie lub działalności.

- *Ontologia zastosowana* przeważnie bazuje na ontologii określającej dziedzinę i ontologii określającej zadanie. Często zdarza się tak, że np. konkretna ontologia dziedziny może być wykorzystywana z różnymi ontologiami określającymi zadania, np. ontologia samochodowych części zamiennych może być wykorzystana razem z ontologią procesu składania nowego samochodu, jak i z ontologią diagnozowania samochodu.

Klasyfikacja ontologii oraz zależności pomiędzy kolejnymi jej grupami zostały przedstawione na rys. 1.1.



Rys. 1.1. Klasyfikacja ontologii [8]

1.3. Zastosowania ontologii

Ontologie szybko zyskały uznanie inżynierów wiedzy oraz badaczy różnych dziedzin nauki. Poniżej opisano kilka przykładowych ontologii z różnych dziedzin, aby pokazać różnorodność branży, w których używa się takiego zapisu wiedzy. Na poniższe przykłady zwrócili również uwagę autorzy książki [5]:

Przykład 1 - The Gene Ontology Project (<http://www.geneontology.org/>)

Zadaniem projektu jest stworzenie słownika pojęć związanych z genetyką dla każdego organizmu. Budowana ontologia składa się z trzech dziedzin: budowa komórki (opisuje elementy występujące w komórce oraz jej otoczenie pozakomórkowe), funkcje molekularne (opis aktywności genów na poziomie molekularnym, np.: procesy wiązania lub katalizy) oraz procesy biologiczne (czynności oraz zestawy wydarzeń na poziomie molekularnym opisane od początku do końca). Ontologia ta jest często aktualizowana i dynamicznie rozwijana. Można ją pobrać ze strony domowej w różnych formatach.

Przykład 2 - The Object-Oriented Software Design Ontology(ODOL)

Ontologia opisuje wzorce projektowe w programowaniu obiektowym. Posiada zbiór pojęć i ich relacji dotyczących programowania obiektowego. Ontologia nie jest zależna od żadnego z języków programowania. Projekt jest próbą przeniesienia wzorców projektowych, które najczęściej są zapisane w nieformalnym języku opisowym lub jako diagramy UML, na język ontologii. W tym przypadku językiem ontologii jest OWL i właśnie w formacie *.owl możemy pobrać ontologię ze strony projektu.

Przykład 3 - The Friend Of A Friend Ontology (FOAF)

Obecnie jedna z najbardziej popularnych ontologii. Ułatwia wymianę informacji o osobach, ich zdjęciach, blogach itp. pomiędzy witrynami sieci web. Powstała przy projekcie FOAF, który zakłada stworzenie sieci gromadzącej strony użytkowników (tzw. sieci społecznej). Strony w tej sieci, dzięki użytej ontologii, mogą być przetwarzane przez komputer.

1.4. Reprezentacja wiedzy

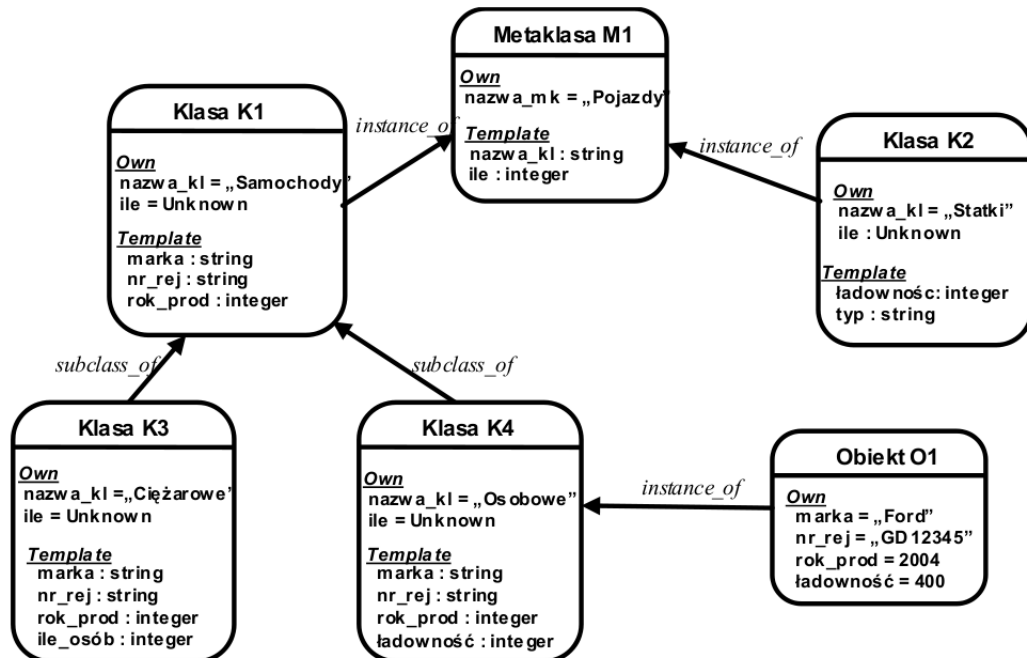
Ramki Minsky'ego

Za jedną z pierwszych ontologicznych metod reprezentacji wiedzy uważa się ramki zaproponowane przez M. Minsky'ego w 1975 roku. Rozwiązanie to opisuje ontologię przy użyciu następujących elementów[7]:

- **klasy** - pojęcia (koncepty),
- **klatki** - własności klas opisujące ich cechy i atrybuty (właściwości, ang. slot). Możemy wyróżnić dwa rodzaje klatek: własne (prywatne własności danej ramki) oraz szablonowe (służą do tworzenia innych ramek, będących wystąpieniami tych pierwszych),
- **fasety** - własności klatek (np. ograniczenia nałożone na klatki),
- **dodatkowe** - ograniczenia wyrażone w języku logiki.

Wedle Minsky'ego elementy otaczającego nas świata możemy opisać za pomocą ramek. Najważniejszymi ramkami są klasy. Klasy posiadają klatki, które mogą być ograniczone przez fasety. Klatki też są ramkami. Każda z klas może posiadać klatki własne i szablonowe. Klasy mogą dziedziczyć od siebie klatki własne, jak i szablonowe, tworząc hierarchię klas. W przypadku wystąpień danej klasy klatki szablonowe

stają się klatkami własnymi. Gdy wystąpienie danej klasy ma klatki szablonowe, to klasę tę nazywamy metaklasą. Jeśli zaś klasa nie ma klatek szablonowych, ani nie może posiadać swoich wystąpień, nazywamy ją obiektem. Przykładowa ontologia zapisana w formie ramek została przedstawiona na rys. 1.2.



Rys. 1.2. Przykładowa ontologia zapisana w formie ramek

Okazało się jednak, że ramki mają kilka wad. Przede wszystkim są zbyt ogólne, nie posiadają ścisłej, formalnej definicji, co sprawiała, że mogą doprowadzić do niejednoznaczności. Wprowadzenie metaklas prowadzi nawet do nierozstrzygalności pewnych problemów wnioskowania.

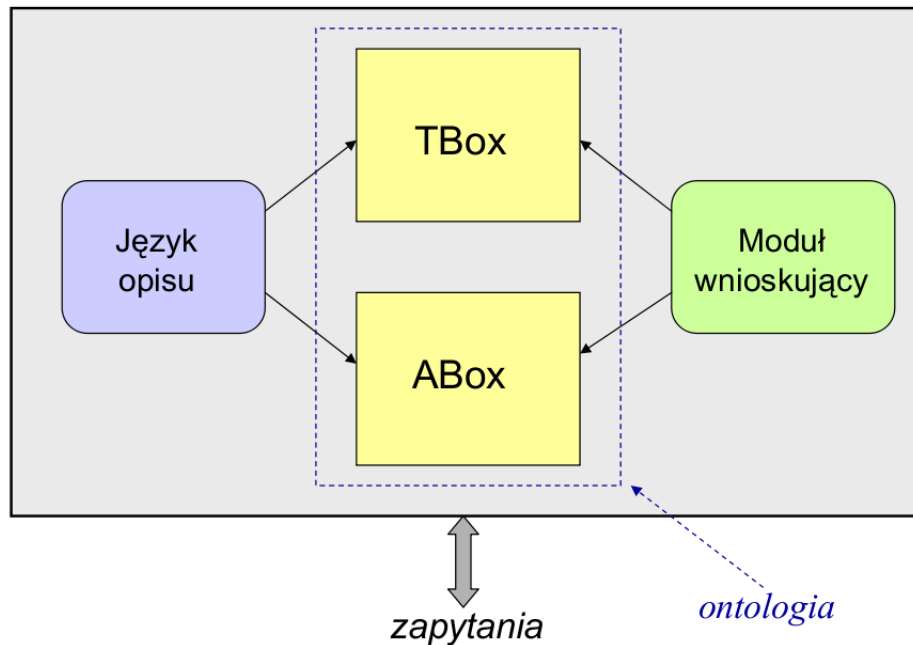
Logika opisowa

Innym językiem reprezentacji wiedzy jest, rozwijana już blisko 30 lat, logika opisowa (ang. Description Logic, DL). Logika opisowa to nie tylko formalny język opisu wiedzy, ale cała rodzina języków przeznaczonych do opisu ontologii, cechujących się różnym stopniem ekspresji oraz możliwości wnioskowania. DL jest formalizmem, opartym na rachunku predykatów pierwszego rzędu, przez co wnioskowanie tak zapisanej wiedzy jest w pełni rozstrzygalne.

Logika opisowa opiera się na prostych i zarazem ogólnych założeniach [7, 6]:

- istnieje pewne *uniwersum*, które chcemy opisać w formie ontologii (dziedzina zainteresowań),

- elementy uniwersum nazywane są osobnikami i są wystąpieniami konceptów,
- elementy ontologii powiązane są binarnymi relacjami.



Rys. 1.3. System zarządzania wiedzą z ontologią DL [6]

Na podstawie powyższych założeń ontologie wyrażone w języku logiki opisowej składają się z dwóch części: TBox (ang. terminological box) i ABox (ang. assertional box). TBox jest zbiorem konceptów, relacji zachodzących pomiędzy nimi oraz zbiorem aksjomatów, które nakładają ograniczenia na koncepty i ich relacje. ABox zawiera wystąpienia pojęć (konceptów) i ról (relacji). Na rysunku 1.3 przedstawiono ogólny model reprezentacji wiedzy w DL.

Pomimo licznej rodziny języków logiki opisowej istnieją wspólne elementy, które posiada każdy z tych języków:

- pojęcia (koncepty) atomowe - koncept uniwersalny \top , który przedstawia naszą dziedzinę zainteresowań oraz koncept pusty \perp , który nie posiada żadnych wystąpień,
- role atomowe,
- konstruktory do tworzenia bardziej złożonych konceptów i ról.

Tab. 1.1. Konstruktory języka ALC [7, 6]

Konstruktor	Znaczenie
$\neg C$	Negacja konceptu
$C \sqcap D$	Cześć wspólna konceptów
$C \sqcup D$	Suma konceptów
$\exists R.C$	Kwantyfikacja egzystencjalna - zbiór osobników powiązanych przynajmniej raz relacją R z osobnikami z konceptu C.
$\forall R.C$	Kwantyfikacja ogólna - zbiór osobników, dla których wszystkie powiązania relacją R odnoszą się do osobników z konceptu C.

W celu dokładniejszego zrozumienia logiki opisowej w tabeli 1.2 została przedstawiona ontologia opisująca rodzinę. Jako język dla zaprezentowanej ontologii wybrano dość prosty, ale pozwalający na definiowanie nietrywialnych ontologii język *ALC*. Konstruktory tego języka zostały przedstawione w tabeli 1.1.

Tab. 1.2. Przykładowa ontologia DL [6]

TBox	ABox
Mężczyzna \sqsubseteq Osoba	Mężczyzna(Karol)
Kobieta \equiv Osoba $\sqcap \neg$ Mężczyzna	Kobieta(Anna)
Rodzic \equiv Osoba $\sqcap \exists$ maDziecko. \top	Kobieta(Joanna)
Ojciec \equiv Mężczyzna \sqcap Rodzic	maDziecko(Anna, Karol)
Matka \equiv Kobieta \sqcap Rodzic	maDziecko(Anna, Joanna)
	maDziecko(Anna, Maria)

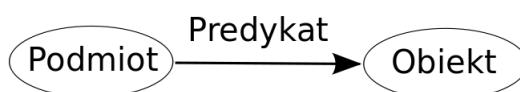
Ontologia rodziny (tabela 1.2) zawiera koncepty: Osoba, Mężczyzna, Kobieta, Rodzic, Matka, Ojciec oraz jedną rolę: maDziecko. TBox przedstawionej ontologii zawiera następujące typy aksjomatów: aksjomat zawierania (oznaczony symbolem \sqsubseteq) oraz aksjomat równoważności (oznaczony symbolem \equiv). Pierwszy aksjomat, będący aksjomatem zawierania, mówi nam, że Mężczyzna jest Osobą. Kolejny przedstawia Kobieta jako Osobę nie będącą Mężczyzną. Następny mówi o tym, że rodzic to osoba, która ma choć jedno dziecko. Ostatnie dwa aksjomaty definiują Ojca i Matkę jak Rodzica będącego odpowiednio Mężczyzną i Kobieta. ABox przedstawionej ontologii zawiera informacje o osobach: Karolu, Annie, Joannie i Marii. Wiemy, że Karol jest mężczyzną, a Anna i Joanna to kobiety. Anna ma troje dzieci: Karola, Joanne i Marię. Płeć Marii nie jest nam znana.

RDF

RDF (ang. Resource Description Framework) jest językiem do opisu zasobów w sieci WWW. Powstał w 1999 roku jak rekomendacja W3C. Język ten opiera się na “trójkowym” modelu danych. Wiedza zapisywana jest za pomocą trójek (triples), które posiadają następujące elementy:

podmiot - predykat - obiekt
podmiot - orzeczenie - dopełnienie
obiekt - właściwość - wartość

Każda trójka może zostać przedstawiona na grafie skierowanym (Rys. 1.4).



Rys. 1.4. Graf przedstawiający trójkę RDF

Dowolny dokument RDF może być przedstawiony za pomocą grafu skierowanego, w którym krawędzie jak i niektóre węzły posiadają URI, czyli ich jednoznaczny identyfikator. W grafie mogą wystąpić węzły puste, które są pomocniczymi węzłami służącymi do opisu wielowartościowych zależności. Węzły puste nie posiadają URI. Podstawowy model zawiera poniższe obiekty[14]:

- *Zasoby* są to dokumenty, części dokumentów, elementy i przedmioty świata rzeczywistego oraz zasoby abstrakcyjne. Na grafie zasoby opisywane są przez węzły identyfikowane przez URI.
- *Właściwości* są zasobami, które opisują właściwości innych zasobów. Na grafie są przedstawiane jako krawędzie skierowane opisane własnym URI.
- *Wyrażenia* są klasycznymi trójkami składającymi się z zasobów, właściwości oraz wartości właściwości. Element trójki będący dopełnieniem może być pewnym zasobem lub wartością literalną. Wartość literalna na grafie jestznaczona jako węzeł oznaczony tą wartością.

Na rysunku 1.5 został zaprezentowany przykładowy graf RDF, który jest prezentacją pliku RDF/XML przedstawionego na listingu 1.1

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
          xmlns:contact="http://www.w3.org/2000/10/swap/pim/contact#">
```

```

<contact:Person rdf:about="http://www.w3.org/People/EM/contact#me">
  <contact:fullName>Eric Miller</contact:fullName>
  <contact:mailbox rdf:resource="mailto:em@w3.org" />
  <contact:personalTitle>Dr.</contact:personalTitle>
</contact:Person>

</rdf:RDF>

```

Listing 1.1. Listing dokumentu RDF opisującego Erica Millera [18]



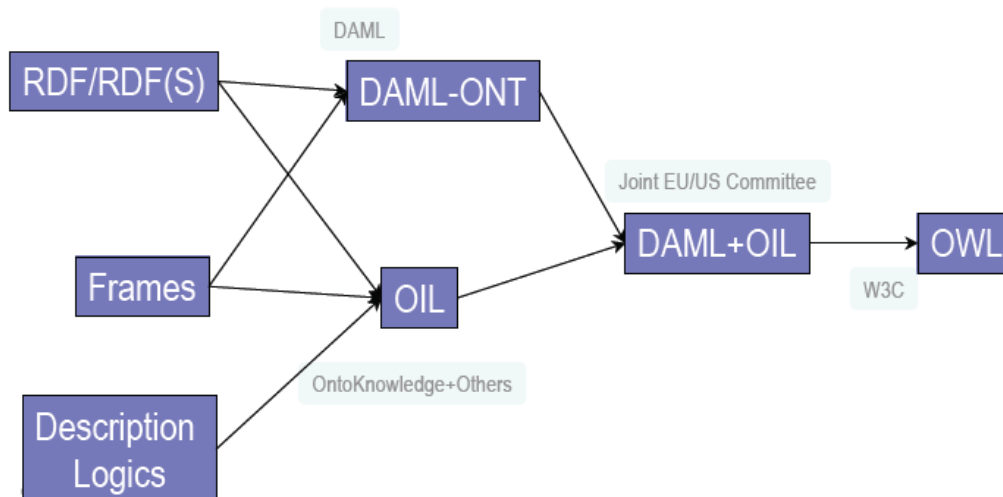
Rys. 1.5. Graf RDF opisujący Erica Millera [18]

Bardzo ważne jest, aby istniała możliwość definiowania typów własnych. Np. typ osoba, który może przechowywać informacje o nazwisku, adresie danej osoby czy jej numerze telefonu. Do tego celu służy RDF Schema.

OWL

Użycie języków RDF i RDF Schema pozawala na tworzenie rozbudowanych ontologii, jednak posiadają one pewne ograniczenia. Nie możemy np. zdefiniować liczby wystąpień danego obiektu. A więc nie będziemy mogli zapisać informacji o tym, że kwartet smyczkowy składa się dokładnie z 4 muzyków.

Powstało jeszcze kilka innych języków opisu ontologii. Jednymi z bardziej dojrzałych są opierające się na wcześniejszych rozwiązaniach języki: DAML i OIL. Język DAML powstał w ramach badań prowadzonych przez wojskową agencję rządową w USA w roku 2000. OIL jest językiem, który powstał przy wsparciu Unii Europejskiej. Pozwala on na definiowanie ontologii, jednak nie posiada wsparcia dla XML. W roku



Rys. 1.6. Drzewo genealogiczne rodziny języków OWL [13]

2003 wyłonił się język będący połączeniem tych dwóch języków: DAML+OIL. Na podstawie tego ostatniego języka W3C stworzyła język OWL (ang. Web Ontology Language). Na rysunku 1.6 przedstawiono historię rozwoju języków zapisu ontologii.

Język OWL stał się standardem zapisu ontologii w sieciach semantycznych. Częściowo bazuje on na trójkach RDF. Semantyka języka OWL opiera się na logice opisowej, a dokładnie na dialekcie SHIOQ(D), który rozszerza konstruktory ALC o np.: ograniczenia liczebności (kardynalność), role przechodnie, symetryczne czy odwrotne.

W celu dostosowania języka OWL do wymogów każdego z użytkowników, posiada on 3 dialekty [24, 1]:

- **OWL Lite** został stworzony z myślą o użytkownikach, którzy chcą zbudować hierarchię i nadać jej niewielkie ograniczenia. OWL Lite pozwala na nadanie kardynalności 0 lub 1. Dialekt ten nie dopuszcza konstrukcji posiadających złożone opisy klas oraz wymaga separacji typów.
- **OWL DL** jest dialektem o dużej sile ekspresji. W szczególności nałożona w nim nacisk na możliwości wnioskowania. OWL DL może być mapowany na logikę opisową SHOIN, przez co mamy pewność, że czas obliczeń wnioskowania jest skończony. Dialekt ten posiada bardzo dużo ograniczeń, np. ograniczenie rozdzielności typów.
- **OWL Full** jest najbardziej rozbudowanym dialektem, w porównaniu do pozostałych dostarczą największej siły ekspresji. Jednak poprzez mniejszą liczbę ograniczeń, w szczególności dotyczących właściwości przechodnich, jest języ-

kiem nierozstrzygalnym. Co oznacza, iż nie mamy pewności, że wnioskowanie zakończy się w skończonym czasie.

Każdy z powyżej opisanych języków jest rozszerzeniem swojego poprzednika. Zachodzi niżej opisana kompatybilność pomiędzy dialektami.

- Każda poprawna ontologia OWL Lite jest poprawną ontologią OWL DL,
- Każda poprawna ontologia OWL DL jest poprawną ontologią OWL Full,
- Każdy poprawny wniosek w OWL Lite jest poprawnym wnioskiem w OWL DL,
- Każdy poprawny wniosek w OWL DL jest poprawnym wnioskiem w OWL Full.

A oto przykład ontologii rodziny zapisany w języku OWL[13]:

Mężczyzna to osoba nie będąca kobietą

```
<owl:Class rdf:ID="Meczczyzna">
  <rdfs:subClassOf rdf:resource="#Osoba" />
  <owl:disjointWith rdf:resource="#Kobieta" />
</owl:Class>
```

Rodzic to taka Osoba, która ma przynajmniej jedno dziecko

```
<!-- definicja klasy rodzic jako klasy rownowaznej
do klasy anonimowej, ktorej wystapienia podlegaja
ograniczeniu liczebnosciowemu (tutaj przynajmniej
1 dziecko) zwiazanemu z wlasciwoscia maDziecko -->
<owl:Class rdf:ID="Rodzic">
  <owl:equivalentClass>
    <!-- klasa rownowazna jest klasa anonimowa o ponizszej
    definicji -->
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Restriction>
          <owl:onProperty rdf:resource="#maDziecko">
            <owl:minCardinality
              rdf:datatype="http://www.w3.org/2001/XMLSchema#int">
                1
            </owl:minCardinality>
```

```

        </owl:Restriction>
        <owl:Class rdf:ID="Osoba"/>
    </owl:intersectionOf>
</owl:Class>
</owl:equivalentClass>
</owl:Class>
<!-- definicja wlasciwosci maDziecko,
ktorej dziedzina i zakresem jest klasa osoba,
wlasciwosc maDziecko jest wlasciwością odwrotna
w stosunku do wlasciwosci maRodzica -->
<owl:ObjectProperty rdf:about="#maDziecko">
    <rdfs:domain rdf:resource="#Osoba"/>
    <rdfs:range rdf:resource="#Osoba"/>
    <owl:inverseOf rdf:resource="#maRodzica" />
</owl:ObjectProperty>
<!-- definicja wlasciwosci maRodzica -->
<owl:ObjectProperty rdf:about="#maRodzica">
    <!-- zdefiniowanie dziedziny i zakresu wlasciwosci,
    dla objectProperty dziedzina i zakres sa opisami klas -->
    <rdfs:domain rdf:resource="#Osoba"/>
    <rdfs:range rdf:resource="#Osoba"/>
    <owl:inverseOf rdf:resource="#maDziecko"/>
</owl:ObjectProperty>

```

Ojciec to Mężczyzna, będący Rodzicem

```

<!-- definicja klasy ojciec jako klasy rownowaznej
do przeciecia (czesci wspolnej) klas rodzic
i mezczyzna -->
<owl:Class rdf:ID="Ojciec">
    <owl:equivalentClass>
        <owl:Class>
            <owl:intersectionOf rdf:parseType="Collection">
                <owl:Class rdf:about="#Rodzic"/>
                <owl:Class rdf:about="#Mezczyzna"/>
            </owl:intersectionOf>
        </owl:Class>
    </owl:equivalentClass>

```



```
</owl:Class>
```

Dziecko to Osoba, która ma dwoje Rodziców

```
<owl:Class rdf:about="#Dziecko">
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Restriction>
          <owl:onProperty>
            <owl:ObjectProperty rdf:ID="maRodzica"/>
          </owl:onProperty>
          <owl:cardinality
            rdf:datatype="http://www.w3.org/2001/XMLSchema#int">
            2
          </owl:cardinality>
        </owl:Restriction>
        <owl:Class rdf:about="#Osoba"/>
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>
```

Język OWL jest obecnie najlepszym sposobem zapisu ontologii, o czym może świadczyć jego popularność wśród inżynierów ontologów. Chociaż umożliwia on zapisanie dowolnej informacji, w październiku 2009 W3C zaproponowało jego ulepszoną wersję OWL2 [21]. Jednak ze względu na wcześniejsze prace grupy projektowej oraz ustalenia z opiekunem projektu, w tej pracy zostanie użyty język OWL.

Rozdział 2

Przegląd dostępnych rozwiązań do wizualizacji ontologii

2.1. Wstęp

W tej części pracy przedstawiona zostanie szczegółowa analiza rozwiązań obrazujących ontologie, która umożliwi wyznaczenie najlepszej aplikacji do wizualizacji.

W pierwszej części rozdziału zostaną opisane sposoby wizualizacji ontologii oraz cechy dobrej wizualizacji. Następnie zostaną przedstawione wybrane aplikacje oraz sposób ich analizy. W ostatniej części rozdziału przedstawione zostaną wyniki testów oraz zostanie wyznaczone najlepsze rozwiązanie.

2.2. Sposoby wizualizacji ontologii

Budowanie ontologii nie jest łatwą pracą. Nawet w niewielkich rozwiązaniach często wymaga analizy dużej liczby elementów. Aby poprawnie tworzyć ontologie, jest wymagana ich wizualizacja. Upraszcza ona postrzeganie całego rozwiązania oraz konkretnych jego części. Umożliwia również łatwą ocenę i poprawę błędów.

Istnieją różne sposoby graficznej prezentacji ontologii. Większość z nich opiera się na modelu 2D, w którym ontologie są wizualizowane za pomocą grafu. Wierzchołki tego grafu oznaczają elementy ontologii, a krawędzie odzwierciedlają związki pomiędzy tymi elementami. Istnieją również rozwiązania 3D np.: OntoSphere [20]. W tym przypadku klasy zostały odzwierciedlone jako sfery (kule), a ich podklasy znajdują się pod nimi w trójwymiarowym drzewie przypominającym stożek. Klasy są połączone trójwymiarową krawędzią o odpowiednim kolorze i kształcie grotu.

2.3. Elementy wizualizacji

Wizualizacja ontologii o dużej liczbie elementów może stać się nieprzejrzysta i nieczytelna, dlatego należy zwrócić uwagę na niżej wymienione elementy [15].

1. Sposób całościowej wizualizacji

Dobra aplikacja do wizualizacji ontologii powinna wyświetlić wszystkie jej elementy. Użytkownik powinien mieć możliwość podglądu całej edytowanej ontologii, jak również wybranej jej części. Aby wizualizacja była przejrzysta i czytelna musi posiadać możliwość wyłączenia wizualizacji niektórych elementów, np. ukrycie rysowania związków danego typu.

2. Sposób wizualizacji klas i bytów

Klasy oraz ich instancje są najważniejszymi elementami ontologii, dlatego jest konieczne ich poprawne wizualizowanie. Zły sposób obrazowania tych elementów może zniechęcić użytkownika do korzystania z rozwiązania. Wizualizacja powinna pokazywać wszystkie klasy lub tylko wybrane przez użytkownika. Każda klasa powinna zawierać przynajmniej nazwę, zapisaną w zrozumiałym sposób. Klasy mogą być obrazowane jako wierzchołki grafu. Powinny one jednoznacznie wyróżniać się od instancji klas. Np. poprzez kolor lub kształt wierzchołka. To podejście może się jednak nie sprawdzić przy wizualizacji ontologii o znacznej liczbie elementów. W takim przypadku może istnieć wiele połączonych z wybranym węzłem klas, obrazowanie takiej sytuacji na grafie może okazać się nieczytelne. Alternatywnym rozwiązaniem jest wyświetlenie wszystkich powiązanych klas z zadany elementem w oddzielnym oknie aplikacji.

3. Wizualizacja taksonomii

Klasy tworzą hierarchię (taksonomię) klas, poprzez relację nadklasa - podklasa (związek isa). Prezentacja taksonomii ma kluczowe znaczenia dla zrozumienia relacji dziedziczenia pomiędzy klasami. Wizualizacja powinna dać możliwość całościowego bądź częściowego przeglądu hierarchii dziedziczenia.

4. Sposób wizualizacji relacji

Relacje występujące pomiędzy elementami są najczęściej obrazowane jako związek łączący te elementy. Różne typy relacji można wyróżnić poprzez nadanie etykiety krawędzi łączącej, poprzez zmianę koloru lub kształtu linii. Ważne jest, aby można było opcjonalnie wyłączyć wizualizację zadanych typów relacji.

5. Wizualizacja właściwości

Ontologie byłyby bardzo ubogie, gdyby nie posiadały właściwości. Właściwości pozwalają na zdefiniowanie ogólnych informacji dotyczących zarówno klas jak i instancji klas. Wyróżniamy dwa typy właściwości: właściwości dla których zakresem są obiekty (`owl:objectProperty`) bądź wartości (`owl:dataProperty`). Właściwości powinny być zaznaczone na grafie wizualizacji lub w oddzielnym oknie do tego przeznaczonym.

6. Wyszukiwanie

Podczas wizualizacji dużej ontologii możemy napotkać na problem szybkiego wyszukania interesującego nas elementu. Dlatego aplikacji wizualizacji powinna dać możliwość szukania elementów w grafie ontologii.

2.4. Przedstawienie wybranych rozwiązań do wizualizacji ontologii

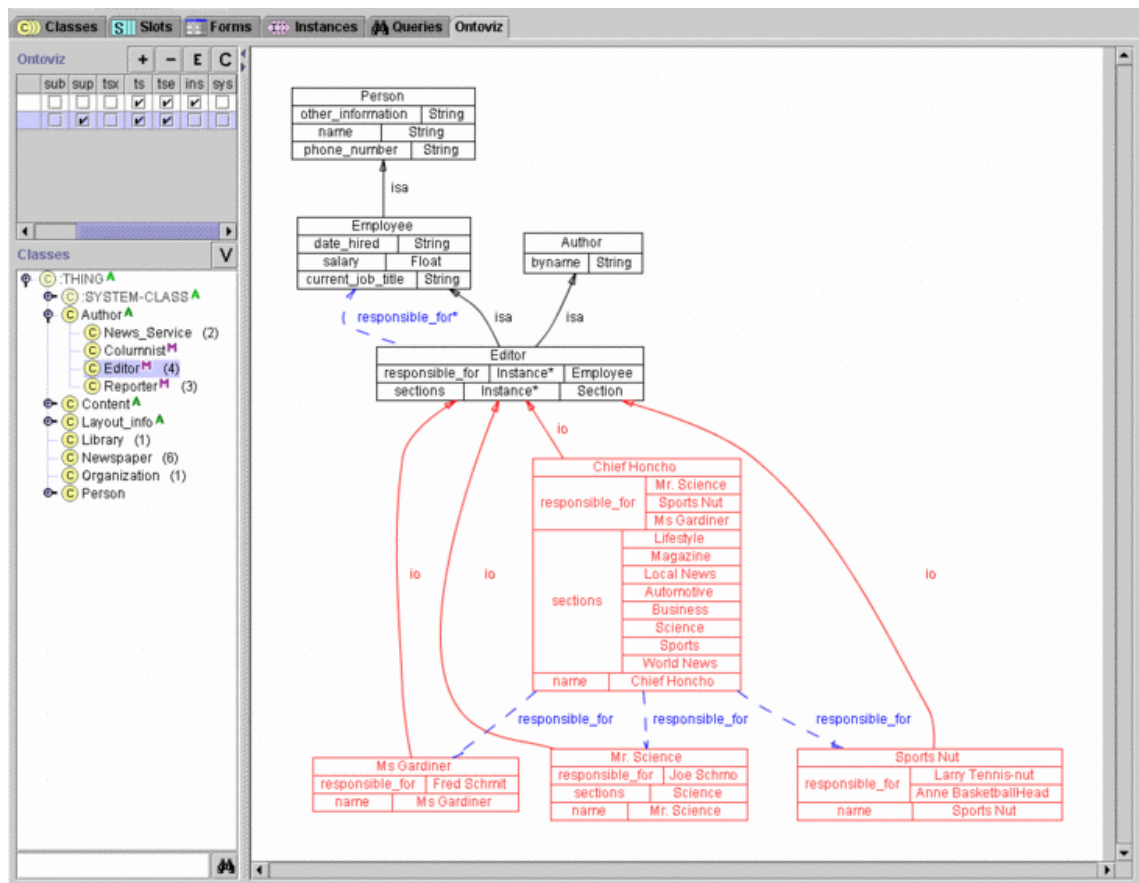
Na rynku dostępnych jest wiele programów do tworzenia i edycji ontologii. Większość z nich ułatwia budowanie ontologii poprzez ich graficzną reprezentację. Niektóre rozwiązanie, takie jak rozwijany na Uniwersytecie Stanforda edytor Protégé, poprzez pluginy dostarczają kilku sposobów wizualizacji. Inne poza ogólnym mechanizmem wizualizacji posiadają dodatkowy tryb do prezentacji drzewa wywnioskowanej hierarchii klas i bytów. Poniżej przedstawiono kilka rozwiązań pozwalających obrazować ontologię.

1. OntoViz

OntoViz [22], rozwijany na Uniwersytecie Stanforda, jest najczęściej używanym pluginem Protégé do wizualizacji ontologii. Wykorzystuje bibliotekę GraphViz do tworzenia prostych grafów 2D (Rys. 2.1). Na grafie klasy są reprezentowane jako prostokąty zawierające informację o nazwie klasy jak i dodatkowe informacje o relacjach i właściwościach klasy. Istnieje możliwość okrojenia i ukrycia części wyświetlanych komponentów przez panel konfiguracyjny po prawej stronie.

2. Jambalaya

Jambalaya [17, 4] jest pluginem do Protégé rozwijanym na Uniwersytecie w Wiktorii (Kanada). Opiera się na graficznym zestawie narzędzi Piccolo do tworzenia interaktywnych grafów 2D. Jambalaya charakteryzuje się kilkoma rodzajami widoku oraz specyficznym sposobem obrazowania związku ISA.



Rys. 2.1. Przykład wizualizacji ontologii za pomocą OntoViz

3. Growl [16]

Rozwiązanie powstało na Uniwersytecie w Vermont. Posiada ono ciekawy sposób wizualizacji ontologii, w którym autorzy postawili na kompletność wizualizacji. Edytor Growl pozwala na wyświetlenie wszystkich elementów ontologii na grafie, co daje możliwość łatwego zrozumienia edytowanej ontologii.

4. OCS

OCS (ang. ONTOLOGY CREATION SYSTEM) [13, 2] jest systemem do tworzenia i edycji ontologii rozwijanym na Wydziale Elektroniki, Telekomunikacji i Informatyki Politechniki Gdańskiej. Edytor ontologii posiada 2 sposoby wizualizacji ontologii. Pierwszym jest hierarchia klas i bytów, drugi zaś to ogólny obraz ontologii. System wizualizuje tylko podstawowe elementy ontologii, przedstawia tylko klasy, instancje i podstawowe relacje pomiędzy nimi.

2.5. Opis metody porównania

Sposób wyznaczenia oceny rozwiązań

Porównanie rozwiązań do wizualizacji ontologii będzie podzielone na kilka kategorii. Każda z kategorii będzie posiadała zestaw zagadnień lub pytań. Odpowiedzi na pytania pozwolą wyznaczyć ocenę dla zadanej kategorii. Każde z zagadnień, poza treścią, posiada również opis badanej cechy, liczbę punktów, które mogą być przyznane za pytanie oraz sposób przydzielania punktów. Ocena dla danej kategorii będzie liczbą zmiennoprzecinkową z przedziału $< 0, 1 >$ i będzie wyznaczana na podstawie wzoru:

$$O_K = \frac{\sum_{k=1}^N p_u(i)}{\sum_{k=1}^N P_{max}(i)} \quad (2.1)$$

gdzie:

N – liczba pytań w kategorii,

O_K – ocena danej kategorii, wartość ta będzie z przedziału $< 0, 1 >$,

$p_u(i)$ – liczba punktów uzyskanych w i -tym pytaniu,

$P_{max}(i)$ – maksymalna liczba punktów możliwa do uzyskania na i -te pytanie.

Całkowita ocena będzie również liczną rzeczywistą z przedziału $< 1, 0 >$, gdzie 1 jest najlepszą notą do zdobycia. Ocena rozwiązania będzie średnią ważoną ocen zdobytych dla poszczególnych kategorii. Wagi zostaną podane poniżej wraz z opisem kategorii.

Opis kategorii

1. Kompletność wizualizacji

Wiele rozwiązań obrazujących ontologie wyświetla tylko podstawowe informacje o klasach i związkach pomiędzy nimi. Aby ułatwić pracę nad ontologiami, wizualizacja musi być kompletna, wszystkie jej elementy muszą być przedstawione na grafie lub w dodatkowych oknach aplikacji. Kryterium to pozwoli ocenić kompletności wizualizacji.

Waga : 5

2. Przejrzystość wizualizacji

Kolejnym kryterium jest przejrzystość wizualizacji. Aby wizualizacja była zrozumiała musi być czytelna, szczególnie gdy wczytamy dużą ontologię. Kryterium to pozwoli sprawdzić, czy aplikacja posiada opcję filtrowania danych i wizualizuje tylko zadane elementy lub związki.

Waga : 4

3. Sposoby wizualizacji

Ontologia może być przedstawiona na drzewie bądź grafie, jej elementy mogą być rozmieszczone w różnych odległościach. Kryterium pozwoli sprawdzić czy poza podstawową wizualizacją rozwiązania posiadają różne algorytmy i sposoby obrazowania danych. Ocenione zostanie również, czy wizualizacja opiera się o dane jawnie pobrane z ontologii, czy może jest wynioskowaną hierarchią klas i bytów.

Waga : 3

4. Użyteczność

Kryterium to pozwoli ocenić w jakim stopniu rozwiązanie jest przyjazne dla użytkownika. Sprawdzi również stopień dostępności pomocy poprzez istnienie strony Internetowej, samouczków czy forów poświęconych danemu rozwiązaniu. Zwrócona zostanie też uwaga na licencję oprogramowania. Wyróżnione zostaną aplikacje wydane na darmowych licencjach i posiadające otwarty kod.

Waga : 3

Kryteria oceny dla kategorii kompletność

Poniżej znajduje się lista kryteriów, które pozwolą na ocenę kompletności wizualizacji. Kryteria zostały wyróżnione i pogrupowane na podstawie elementów składowych ontologii.

Kryterium:	Sposób wizualizacji klas
Opis kryterium:	Klasy są najważniejszymi elementami ontologii, dlatego wizualizacja powinna zawierać przynajmniej nazwę klasy. Klasy poprzez kolor lub kształt powinny wyróżniać się na tle innych elementów ontologii.
Liczba punktów do uzyskania:	2
Sposób oceny:	1 punkt za wizualizację i jednoznaczne oznaczenie klas, 1 punkt za wyróżnienie klas od innych elementów ontologii.

Kryterium:	Sposób wizualizacji bytów
Opis kryterium:	Byty, będące instancjami klas, powinny odróżniać się od klas, np. poprzez kolor lub kształt wierzchołka. Każdy z nich powinien identyfikować się nazwą.

Liczna punktów do uzyskania:	2
Sposób oceny:	1 punkt za wizualizację i odpowiednie oznaczenie bytów, 1 punkt za wyróżnienie bytów od innych elementów np. klas.

Kryterium:	Sposób wizualizacji klas anonimowych
Opis kryterium:	Klasy anonimowe są rezultatem relacji innych klas. Klasą anonimową może być np. suma dwóch klas. Elementy te nie posiadają nazwy, dlatego wizualizacja powinna pokazać z jakiej relacji i z jakich klas one powstały.
Liczna punktów do uzyskania:	2
Sposób oceny:	1 punkt za wizualizację klas anonimowych, 1 punkt za przejrzyste ukazanie relacji i klas, których owa klasa jest rezultatem

Kryterium:	Wizualizacja relacji dotyczących klas i bytów
Opis kryterium:	Kryterium pozwoli ocenić stopień pokrycia związków pomiędzy klasami i bytami
Liczna punktów do uzyskania:	11
Sposób oceny:	1 punkt za wizualizację związku "subclass", 1 punkt za wizualizację związku "instanceOf", 1 punkt za wizualizację związku "equivalentClass", 1 punkt za wizualizację związku "disjointWith", 1 punkt za wizualizację związku "differentFrom", 1 punkt za wizualizację związku "allDifferent", 1 punkt za wizualizację związku "sameAs", 1 punkt za wizualizację związku "oneOf", 1 punkt za wizualizację związku "unionOf", 1 punkt za wizualizację związku "intersectionOf", 1 punkt za wizualizację związku "complementOf"

Kryterium:	Wizualizacja właściwości
Opis kryterium:	Właściwości pozwalają na zdefiniowanie ogólnych informacji dotyczących zarówno klas jak i instancji klas.
Liczna punktów do uzyskania:	2
Sposób oceny:	1 punkt za wizualizację DataTypeProperty, 1 punkt za wizualizację ObjectProperty,

Kryterium:	Wizualizacja związków odnoszących się do właściwości
Opis kryterium:	Kryterium pozwoli ocenić stopień pokrycia związków dotyczących właściwości
Liczna punktów do uzyskania:	7
Sposób oceny:	1 punkt za wizualizację związku "subProperty", 1 punkt za wizualizację związku "equivalentProperty", 1 punkt za wizualizację związku "functionalProperty", 1 punkt za wizualizację związku "inversFunctionalProperty", 1 punkt za wizualizację związku "symmetricProperty", 1 punkt za wizualizację związku "transitiveProperty", 1 punkt za wizualizację związku "inverseOf(property)",

Kryterium:	Wizualizacja pozostałych elementów ontologii
Opis kryterium:	Kryterium pozwoli ocenić stopień pokrycia pozostałych elementów ontologii, takich jak np. kardynalność.
Liczna punktów do uzyskania:	4
Sposób oceny:	1 punkt za wizualizację "sameValuesFrom/allValuesForm", 1 punkt za wizualizację "Cardinality", 1 punkt za wizualizację "Domain", 1 punkt za wizualizację "Range",

Kryteria oceny dla kategorii przejrzystość wizualizacji

Poniżej znajduje się lista kryteriów i pytań, które pozwolą na ocenę czytelności i zrozumiałości wizualizacji.

Kryterium:	Czytelność ontologii
Opis kryterium:	Kryterium pozwoli ocenić jakość wizualizacji dużej, liczącej ponad 100 klas ontologii. Wizualizacja powinna dać możliwość wyświetlenia całej ontologii. Oceniony zostanie sposób rozmieszczenia danych. Należy sprawdzić, czy elementy nie nachodzą na siebie i czy nie zakrywają innych elementów ontologii.
Liczba punktów do uzyskania:	5
Sposób oceny:	0-3 punktów za ogólną czytelność dużej ontologii. 0-2 punktów za nienachodzenie na siebie elementów.

Kryterium:	Dostępność filtrów
Opis kryterium:	Wprowadzenie możliwości wizualizacji tylko wybranych typów elementów ontologii znacznie ułatwia pracę z tą ontologią. Wizualizacja powinna dać możliwość filtrowania danych. Kryterium to sprawdzi jakie filtry posiadają wybrane rozwiązania i w jakim stopniu filtry pokrywają liczbę typów wyświetlanych elementów.
Liczba punktów do uzyskania:	3
Sposób oceny:	0-3 punkty za ilość filtrów.

Kryteria oceny dla kategorii sposoby wizualizacji

Poniżej zamieszczono listę kryteriów, które pozwolą ocenić różnorodność sposobów wizualizacji zaproponowanych w testowanych rozwiązaniach.

Kryterium:	Liczba sposobów wizualizacji
Opis kryterium:	Kryterium pozwoli sprawdzić, ile różnych sposobów rozmieszczenia danych i wizualizacji znajduje się w danym rozwiązaniu.
Liczba punktów do uzyskania:	3

Sposób oceny:	0 punktów, jeśli rozwiązanie posiada tylko jeden sposób wizualizacji 1 punkt, jeśli wizualizacja posiada 2 różne sposoby obrazowania danych 2 punkty dla 3 różnych sposobów wizualizacji 3 punkty, jeśli rozwiązanie pozwala na wizualizację za pomocą więcej niż 3 różnych algorytmów obrazowania danych
---------------	--

Kryterium:	Wizualizacja wywnioskowanej hierarchii
Opis kryterium:	Wywnioskowana hierarchia bytów i klas pozwala na łatwiejsze zrozumienie ontologii. Może być też przydatna przy kontroli jakości ontologii, np. przy sprawdzaniu spójności ontologii.
Liczna punktów do uzyskania:	1
Sposób oceny:	1 punkt, jeśli rozwiązanie umożliwia wizualizację hierarchii wygenerowanej przez narzędzie wnioskujące

Kryteria oceny dla kategorii użyteczność

Poniżej znajduje się lista kryteriów które pozwolą ocenić użyteczność rozwiązań.

Kryterium:	Użyteczność
Opis kryterium:	Oceniona zostanie intuicyjność łatwość korzystania i rozwiązania.
Liczna punktów do uzyskania:	6
Sposób oceny:	0-2 punktów za łatwość instalacji 0-3 punktów za intuicyjność interfejsu 1 punkt za możliwość wyszukiwania

Kryterium:	Dostępność pomocy
Opis kryterium:	Pozwala określić stopień dostępności pomocy dla rozwiązania
Liczna punktów do uzyskania:	3

Sposób oceny:	1 punkt dla rozwiązań posiadających własną stronę internetową 1 punkt za istnienie samouczków lub instrukcji użytkowania dla rozwiązania 1 punkt za dostępność forów związanych z rozwiązaniem.
---------------	---

Kryterium:	Licencja
Opis kryterium:	Zostanie sprawdzona licencja, na której zostało wydane oprogramowanie.
Liczba punktów do uzyskania:	2
Sposób oceny:	1 punkt, jeśli oprogramowanie jest na darmowej licencji 1 punkt, jeśli posiada otwarty kod.

2.6. Test rozwiązań do wizualizacji ontologii

Wyniki przeprowadzonych badań (zawarte w Tab. 2.15) ukazują, iż na rynku nie ma idealnego rozwiązania do wizualizacji ontologii. Żadna z badanych aplikacji nie uzyskała wyniku większego niż 70% maksymalnej oceny. Najlepszym rozwiązaniem okazała się Jambalaya, która uzyskała notę 0.66. Jednak poza oceną całościową należy zwrócić uwagę na oceny w poszczególnych kategoriach. Stopień pokrycia wizualizowanych elementów dla Jambalaya'i wynosi tylko 40%. W tej kategorii bardzo dobrze wypadł Grawl, który wizualizuje większość, bo aż 77% elementów.

Ogólnie niski poziom narzędzi do wizualizacji ontologii jest motywacją na stworzenia nowego rozwiązania, które łączyłoby by wszystkie najlepsze cechy badanych edytorów. Takie narzędzie mogłoby zastąpić obecny moduł wizualizacji ontologii używany w katedralnym systemie OCS.

Tab. 2.15. Porównanie rozwiązań do wizualizacji ontologii

Kryterium	OntoVis	Jambalaya	Growl	OCS
Kompletność wizualizacji (max 30 punktów)				
Sposób wizualizacji klas	2	2	2	2
Sposób wizualizacji bytów	2	2	2	0
Wizualizacja klas ananimowych	0	0	2	0
Wizualizacja relacji	4	3	10	2
Wizualizacja właściwości	2	2	2	0
Wizualizacja związków odnoszących się do właściwościami	0	0	4	0
Wizualizacja pozostałych elementów ontologii	0	3	1	0
Łącznie dla kategorii	0,33	0,4	0,77	0,13
Przejrzystość wizualizacji (max 8 punktów)				
Czytelność ontologii	3	4	1	2
Dostępność filtrów	2	3	0	0
Łącznie dla kategorii	0,625	0,875	0,125	0,25
Sposoby wizualizacji (max 4 punkty)				
Liczba sposobów wizualizacji	0	3	1	1
Wizualizacja wywnioskowanej hierarchii	0	0	0	1
Łącznie dla kategorii	0	0,75	0,25	0,5
Użyteczność (max 11 punktów)				
Użyteczność	4	5	5	5
Dostępność pomocy	2	2	2	0
Licencja	2	1	2	2
Łącznie dla kategorii	0,73	0,73	0,82	0,64
Ocena rozwiązania	0,42	0,66	0,5	0,34

Rozdział 3

Analiza i projekt systemu

3.1. Wstęp

W tej części pracy przedstawiona zostanie analiza systemowa i projekt techniczny tworzonej aplikacji do wizualizacji ontologii. Część prac projektowych oraz implementacyjnych została wykonana w czteroosobowym zespole w ramach projektu grupowego. Cele, wymagania i założenia zostały przedyskutowane z opiekunem pracy oraz innymi pracownikami Katedry Architektury Systemów Komputerowych. W ramach tych dyskusji ustalono, że tworzona aplikacja zostanie wydana jako biblioteka, którą programiści będą mogli wdrożyć w swoich rozwiązaniach.

W pierwszej części tego rozdziału zaprezentowane zostaną cele i wymagania stawiane tworzonej aplikacji. Następnie przedstawione zostaną aspekty techniczne dotyczące bibliotek pomocniczych, które zostaną wykorzystane w rozwiązaniu. Na podstawie tej wiedzy zostanie przeprowadzona analiza SWOT oraz określony zostanie harmonogram prac. W ostatniej części rozdziału zostanie zaprezentowany diagram pakietów. Każdy z pakietów zostanie krótko omówiony. Ze względu na zbyt dużą liczbę klas nie zostanie tutaj zaprezentowana pełna analiza obiektowa, która została przeprowadzona podczas realizacji projektu.

3.2. Cele systemu

Głównym celem systemu jest stworzenie biblioteki pozwalającej na wizualizację ontologii zapisanych w języku OWL. Istnieje zapotrzebowanie na bibliotekę, która tłumaczyłaby OWL bezpośrednio na elementy graficzne. Programiści aplikacji związanych z ontologiami korzystając z gotowej biblioteki do wizualizacji, będą mogli więcej czasu poświęcić innym zagadnieniom tworzonej przez nich aplikacji.

Aby przetestować rozwiązanie zostanie ono wdrożone do rozwijanego przez Katedrę Architektury Systemów Komputerowych systemu OCS [13, 2]. Na podstawie przeprowadzonych testów wiadomo, że moduł wizualizujący ontologię w systemie OCS wymaga modernizacji i rozbudowy funkcjonalności. Biblioteka wizualizująca ontologię ułatwi i przyspieszy zakończenie projektu OCS. Pozwoli również na zwiększenie atrakcyjności portalu OCS i przyciągnięcie użytkowników.

Zostanie również stworzony plugin do aplikacji Protégé, dzięki niemu bibliotek trafi do większego grona użytkowników. Ponieważ autorzy Protégé dokonali zmian w architekturze ich edytora, wcześniej stworzone pluginy przestały działać w nowej wersji aplikacji. Dlatego istnieje zapotrzebowanie na plugin wizualizujący, który pozwoli na zwiększenie efektywności tworzenia ontologii.

3.3. Użytkownicy systemu

Ze względu na specyfikę projektu można wyróżnić dwa rodzaje użytkowników: programistę i twórcę ontologii.

1. **Programista** jest użytkownikiem, który wykorzystuje stworzoną bibliotekę w swoim rozwiązaniu. Użytkownik ten będzie wymagał od biblioteki łatwości użycia, intuicyjności oraz dobrej dokumentacji. Programista jest użytkownikiem, który może nie znać języka OWL. Jego zadaniem jest stworzenie aplikacji, która pozwoli m.in. obrazować ontologie.
2. **Twórca ontologii** jest użytkownikiem, który zajmuje się tworzeniem i edycją ontologii. Będzie on korzystał z systemu OCS lub pluginu do Protégé. Twórca ontologii najczęściej jest specjalistą w dziedzinie ontologii oraz dobrze zna zagadnienia związane z językiem OWL.

3.4. Wymagane funkcje systemu

Poniżej zostały przedstawione najważniejsze wymagania stawiane tworzonej bibliotece. Na liście znalazły się m.in. wymagania funkcjonalne i wymagania jakościowe.

1. Biblioteka powinna udostępniać kilka trybów prezentacji grafów (np. w formie drzewa, w formie gwiazdy i innych).
2. Biblioteka powinna dać możliwość wizualizacji wywnioskowanej hierarchii klas i bytów.

3. Domyślne parametry w trybach wizualizacji (takie jak długość krawędzi grafu, automatyczne układanie) powinny zostać dobrane w taki sposób, by obraz był przejrzysty, stabilny i czytelny.
4. Rozróżnianie podstawowych symboli. Class, Individual, Property powinny mieć wyróżniające je symbole.
5. Rozróżnianie szczególnych typów Class. Klasa anonimowa, datatype, Thing i Nothing powinny być łatwo rozpoznawalne.
6. Rozróżnianie związków między klasami (Class), bytami (Individual) oraz właściwościami (Property). Różne symbole dla `equivalentClass`, `disjointWith`, `subClassOf`, `sameAs`, `differentFrom`, `allDifferent`, `oneOf`, `unionOf`, `intersectionOf`, `complementOf`, `subProperty`, `equivalentProperty`, `hasProperty`.
7. Rozróżnianie ograniczeń predykatów (Restrictions). Wyróżnić należy kardynalność (`cardinality`), domeny (`domains`) predykatów, `inverseOf`, właściwości predykatów (`transitive`, `symmetric`, `functional`, `inverseFunctional`).
8. Powinna istnieć możliwość wyszukiwania elementów ontologii na wyświetlanym grafie. Elementy spełniające kryterium wyszukiwania powinny zostać wyróżnione.
9. Każdy z obrazowanych elementów powinien mieć możliwość zmiany jego koloru, tak aby użytkownik mógł dostosować wizualizację do swoich potrzeb.
10. Wszystkie wizualizowane elementy powinny pochodzić z ontologii otrzymanej na wejściu programu. Program nie powinien dodawać własnych elementów (np. wywnioskowanych). Wyjątkowo dla klas, które nie mają zdefiniowanych nadklas zostanie utworzony związek z klasą `Thing`.
11. Biblioteka będzie udostępniać strumień danych, w którym znajdą się komunikaty o błędach. Strumień ten będzie mógł zostać wykorzystany przez użytkownika.
12. Jeżeli biblioteka nie wizualizuje danej funkcji OWL API, informacja o tym powinna znaleźć się w strumieniu błędów.

3.5. Nazwa i logo tworzonej biblioteki

Po chwili burzliwych przemyśleń w obrębie zespołu postanowiono, iż biblioteka będzie posiadać nazwę: SOVA, z ang. Simple Ontology Visualization API (co można

przetłumaczyć na polski jako: proste API do wizualizacji ontologii). Fonetyczna wymowa nazwy biblioteki oznacza ptaka - sowę. Sowa w języku angielskim to owl, co oczywiście kojarzy się z językiem OWL i tu koło znaczeń się zamyka.

Jako logo biblioteki wybrano właśnie sowę. Logo zostało zaprezentowane na rysunku 3.1. Rysunek został pobrany z portalu Clker.com [www.clker.com], gdzie został upubliczniony na licencji: creative common public domain license. Licencja umożliwia darmowe kopiowanie, modyfikowanie i upublicznianie pobranej grafiki.



Rys. 3.1. Logo tworzonej biblioteki

3.6. Wybór technologii

Tworzona aplikacja zostanie napisana w Javie 1.6. O wyborze języka zdecydował fakt, iż zarówno OCS, jak i Protégé są napisane w tym języku.

3.6.1. Wybór biblioteki graficznej

Napisanie całego kodu pozwalającego na wizualizację byłoby czasochłonne i wymagało dużego nakładu pracy. Dlatego aplikacja będzie się opierać na istniejącym już rozwiązaniu, pozwalającym na obrazowanie danych za pomocą grafów. W sieci można odnaleźć dużo takich rozwiązań. Jednak użyta biblioteka obrazująca grafy musi być nieodpłatna i najlepiej posiadać otwarty kod. Poniżej przedstawiono najciekawsze rozwiązania, które były brane pod uwagę przy wyborze biblioteki.

Prefuse [10, 19] jest elastycznym pakietem dostarczającym programiście narzędzia do przechowywania danych, manipulowania nimi oraz ich interaktywnej wizualizacji. Biblioteka jest rozwijana w całości w języku Java. Może być wykorzystana do budowania niezależnych aplikacji, wizualnych komponentów rozbudowanych aplikacji oraz tworzenia apletów.

Podstawowe cechy i elementy biblioteki Prefuse:

- różne algorytmy i metody wizualizacji danych m.in.: ForceDirectedLayout, RadialTreeLayout, NodeLinkTreeLayout, SquarifiedTreeMapLayout,
- dynamiczne rozmieszczanie i animacje,
- transformacje, przekształcenia geometryczne oraz przybliżanie/oddalanie obrazu,
- podstawowym elementem struktury danych jest krotka,
- krotki mogą być tworzone bezpośrednio w aplikacji lub na podstawie zewnętrznych danych,
- wbudowany język zapytań do filtrowania danych,
- tworzenie struktur danych na podstawie zewnętrznych plików (CSV, XML) oraz bazy danych,
- klasy wspomagające synchronizację danych pomiędzy tabelami Prefuse, a bazą danych,
- Prefuse posiada licencję BSD.

Piccolo jest zastawem narzędzi używanych przy tworzeniu graficznych aplikacji, często wykorzystywanym do tworzenia interfejsów użytkownika, w których elementy są przybliżane i oddalane. Istnieją trzy wersje tej biblioteki: Piccolo.Java, Piccolo.NET oraz PocketPiccolo.NET. Posiada Licencję BSD.

JUNG (Java Universal Network/Graph Framework) Biblioteka przeznaczona do wizualizacji danych za pomocą grafów oraz sieci. Umożliwia wizualizację nie tylko grafów prostych, ale m.in. multigrafów, digrafów oraz grafów posiadających wagi i etykiety na wierzchołkach i krawędziach. Biblioteka posiada podstawowe algorytmy grafowe. Została napisana w całości w Javie i wydana na licencji BSD.

JGraph Napisana w pełni w Javie biblioteka do wizualizacji grafów kompatybilna ze Swingiem. Posiada wiele ciekawych opcji wizualizacji zarówno wierzchołków, jak i krawędzi grafów. Poza algorytmami wizualizacji w jej skład wchodzi podstawowe algorytmy grafowe. Została wydana na licencji LGPL.

Po uważnym przejrzaniu bibliotek, najbardziej użyteczne wydają się Prefuse oraz Piccolo. Ze względu na dostępność dużej liczby przykładowego kodu wykorzystującego Prefuse w edytorze OCS, wykorzystana zostanie biblioteka Prefuse. Ponadto opinie wyrażone w pracy magisterskiej Andrzeja Jakowskiego[13] silnie przemawiają na korzyść Prefuse.

3.6.2. Format danych wejściowych

Celem aplikacji jest wizualizowanie ontologii zapisanych w języku OWL. Za język danych, które mają zostać zobrazowane, przyjęto OWL DL, który jest wykorzystywany w systemie OCS. Nie oznacza to jednak, że pozostałe dialekty nie będą obsługiwane, jednak w OWL Full możemy napotkać na pewne niejasności (szczególnie pod względem rozróżniania typów).

W celu wczytania ontologii z pliku, zostanie wykorzystana biblioteka OWL API [11, 12]. Ontologie, zapisane jako obiekty OWL API, będą danymi wejściowymi dla biblioteki SOVA. Co oznacza, że dla biblioteki wizualizującej ontologie nie jest istotne źródło danych (plik, baza danych). Aby zachować kompatybilność z systemem OCS zostanie użyta biblioteka OWL API w wersji 2.1.1.

3.7. Planowanie projektu

Planowanie jest częścią projektu, którego zadaniem jest osiągnięcie celu projektu z uwzględnieniem jego ograniczeń. Poniżej zostanie przedstawiony harmonogram prac oraz analiza SWOT, które pozwolą lepiej zaplanować i zrozumieć tworzony projekt.

3.7.1. Harmonogram projektu



Nazwa zadania	Data rozpocz...	Data zakończ...
Analiza obecnego rozwiązania	25.01.10	19.02.10
Implementacja filtrów	22.02.10	19.03.10
implementacja wywnioskowanej hierarchii	22.03.10	20.04.10
Implementacja widoku RadialTree	22.04.10	19.05.10
Plugin do protege	21.05.10	02.07.10
Wdrożenie do OCS	09.07.10	31.07.10
Przegląd literatury	25.01.10	25.03.10
Rozdział 2 - porównanie istniejących rozwiązań	08.02.10	06.03.10
Rozdział 3 - projekt systemu	08.03.10	01.05.10
Rozdział 4 - implementacja	03.05.10	21.07.10
Rozdział 5 - testy aplikacji	26.07.10	27.08.10
Rozdział 6 - Podsumowanie projektu	31.08.10	11.09.10

Rys. 3.2. Harmonogram projektu

Napisanie pracy dyplomowej wiąże się z napisaniem projektu informatycznego (aplikacji) oraz formalnym opisem pracy w postaci tego dokumentu. Na rysunkach 3.2 i 3.8 przedstawiony został harmonogram, w którym uwzględniono prace związane z formalnym opisem projektu, jak i jego implementacją.

3.7.2. Analiza SWOT

W tabeli 3.1 przedstawiono analizę SWOT przeprowadzoną dla tworzonego projektu. Pozwoliła ona na wyróżnienie słabych i mocnych stron projektu oraz dała możliwość poznania szans, dzięki którym projekt może zakończyć się powodzeniem, i zagrożeń, które mogą spowodować, iż projekt nie zostanie zaakceptowany przez rynek i użytkowników.

Tab. 3.1. Analiza SWOT

Mocne strony	Słabe strony
<ol style="list-style-type: none">1. Dobra znajomość języka Java wśród członków zespołu.2. Znajomość zagadnień związanych z ontologiami.3. Współpraca z pracownikami Katedry Architektury Systemów Komputerowych.	<ol style="list-style-type: none">1. Brak doświadczenia w pracy z biblioteką Prefuse.2. Brak doświadczenia w pracy z biblioteką OWL API.3. Problem z uzyskaniem zgody Politechniki Gdańskiej na upublicznienie aplikacji.
Szanse	Zagrożenia
<ol style="list-style-type: none">1. Wzrost zainteresowania sieciami semantycznymi i ontologiami.2. Brak dobrych rozwiązań do wizualizacji ontologii, co sprawia, że istnieje zapotrzebowanie na bibliotekę do wizualizacji ontologii.	<ol style="list-style-type: none">1. Zmniejszenie zainteresowania językiem OWL na korzyść OWL 2.2. Wprowadzenie nowego, lepszego sposobu zapisu wiedzy.3. Konkurencja ze strony istniejących rozwiązań, które umożliwiają wizualizację ontologii.

3.8. Projekt Wizualizacji

Specyfikacja języka OWL pozwala na dużą dowolność definiowania ograniczeń czy opisów elementów ontologii. Dlatego wizualizacja powinna odzwierciedlać elementy ontologii w sposób opisany przez autora w języku OWL. Ważne jest, aby

graficzna reprezentacja ontologii była jednoznaczna i zarazem łatwa do zrozumienia. Chcąc spełnić te kryteria należy zdefiniować sposób obrazowania każdego z elementów języka OWL DL. Źle zaprojektowana wizualizacja może być przyczyną niepowodzenia projektu.

Najważniejszymi elementami ontologii są klasy, właściwości, typy danych oraz byty. Trzy pierwsze elementy biorą udział w podobnych związkach, dlatego będą obrazowane w postaci zaokrąglonego prostokąta o kolorze zależnym od rodzaju elementu. Reprezentacją bytów, aby wyróżnić je od pozostałych elementów, będzie prostokąt. Byty będą posiadały też inny kolor niż pozostałe elementy. (Rys. 3.3).

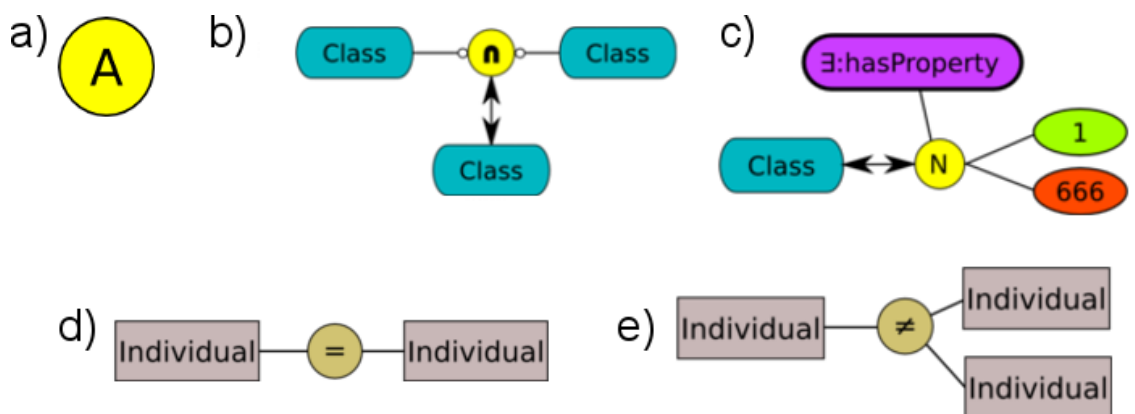


Rys. 3.3. Symbole reprezentujące klasę (a), właściwość (b), typ danych (c) oraz byt (d)

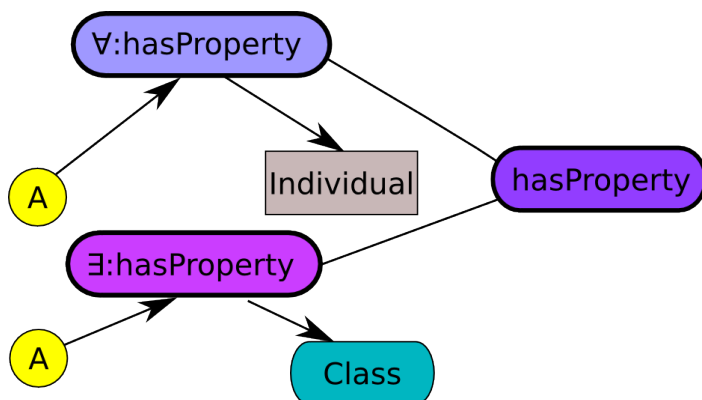
Największym wyzwaniem wizualizacji jest zrozumiałe przedstawienie klas anonimowych. Zaproponowano symbole przedstawiające złożone relacje, w postaci kółek z wpisanymi w nie znaczącymi symbolami. Przykładowe symbole zaprezentowano na Rys. 3.4. Wykorzystano symbolikę matematyczną w przypadku intersekcji, komplementarności, unii oraz kardynalności. Matematyczną symbolikę otrzymały także relacje zachodzące pomiędzy bytami - „sameAs” i „allDifferent”, przy czym symbol relacji „sameAs” jest nadmiarowy, aby zachować spójność wizualizacji (relacja ta jest przeciwna do differentFrom/allDifferent). Kardynalność również reprezentowana jest za pomocą anonimowego wierzchołka, przy czym dodatkowy wierzchołek z ograniczeniem liczby jest wyróżniony kolorem w zależności od typu ograniczenia (min, max, equal)

Zaprezentowanie relacji „allValuesFrom” i „someValuesFrom” odbywa się poprzez wprowadzenie klasy anonimowej reprezentującej wynik podanego ograniczenia (Rys. 3.5). Jest ona połączona z symbolem właściwości, przy czym nazwa właściwości, występująca w aksjomacie, poprzedzona jest kwantyfikatorem ogólnym dla „allValuesFrom” oraz szczególnym dla „someValuesFrom”. Następnie strzałka wskazuje klasę (warto zwrócić uwagę, że może to być także dowolnie złożona klasa anonimowa), określającą przeciwdziedzinę przedstawionej relacji. Dodatkowo użyty wierzchołek property połączony jest z wierzchołkiem przedstawiającym jego definicję.

Różne proste relacje reprezentowane są przez strzałki o różnych grotach. Relację „SubClass” i „SubProperty” prezentuje zaczerpnięta ze specyfikacji UML - strzałka o pustym grocie. Krawędzie reprezentujące związki „equivalent” i „disjoint” mają odwrotne groty, które podkreślają odwrotność tych relacji.

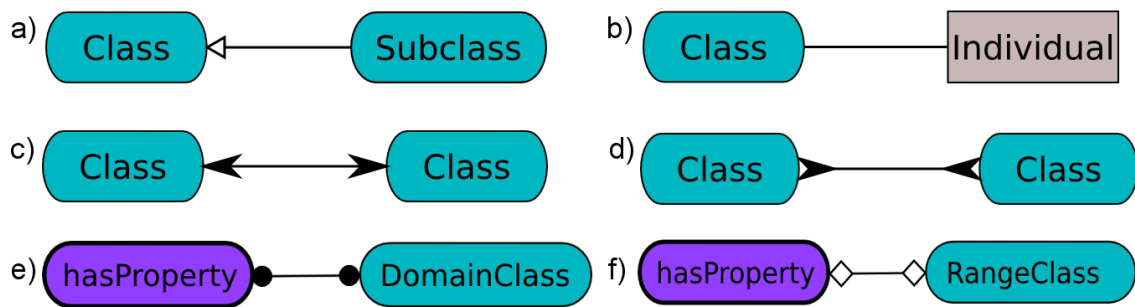


Rys. 3.4. Przykładowe symbole reprezentujące klasę anonimową (a), intersekcję (b), kardynalność typu max i min (c), relację sameAss (d) oraz relację allDifferent (e)



Rys. 3.5. Reprezentacja relacji someValuesFrom oraz allValuesFrom

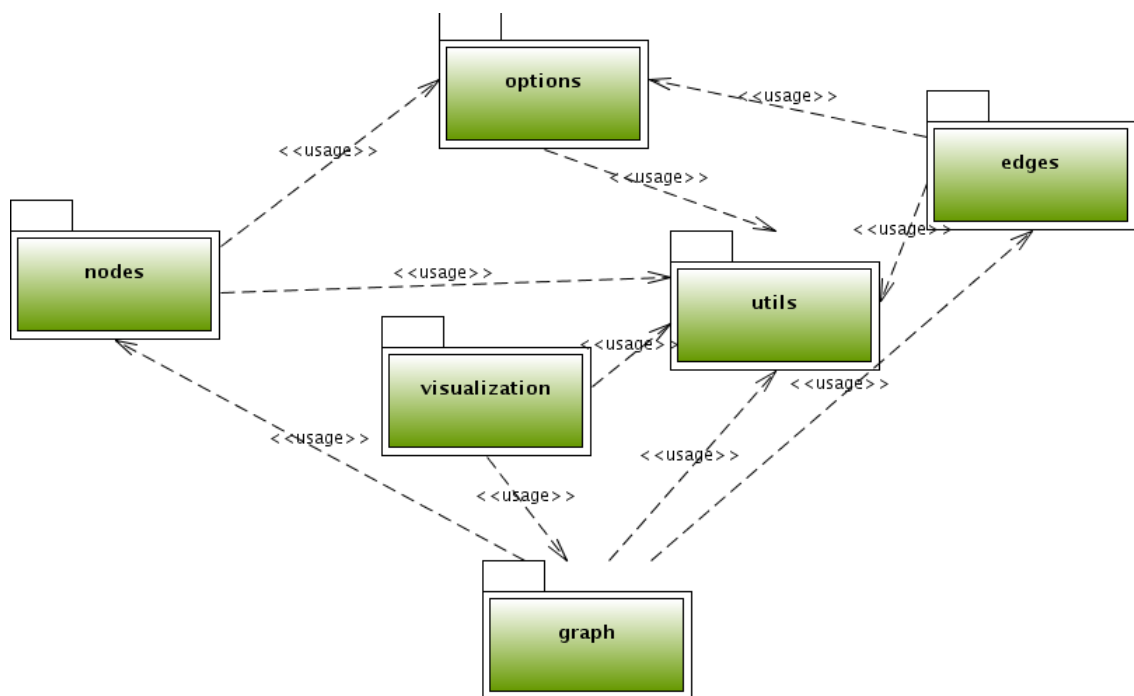
W przypadku definicji właściwości (Property), inwersja jest wyróżniona kolorem czerwonym i oznaczana dwójako, ze względu na asymetryczność tej relacji. Z kolei równoważność właściwości odróżniona jest od równoważności klas kolorem. Zastosowane zostały dodatkowe rodzaje grotów, umożliwiające łatwe rozróżnienie range i domain dla property. Do oznaczenia konkretnych property wykorzystane zostały tradycyjne strzałki, które wskazują kierunek czytania aksjomatu. Aksjomaty, które tworzą kolejne klasy anonimowe, są czytelne dzięki specjalnemu grotowi, który (podobnie jak przy właściwościach) był konieczny, aby wskazać kierunek ich czytania. Tam, gdzie nie jest to potrzebne, krawędzie nie posiadają grotów, co zwiększa czytelność wizualizacji. Przykładowe relacje zaprezentowano na Rys. 3.6.



Rys. 3.6. Przykładowe symbole reprezentujące relacje proste: `rdfs:subclassOf` (a), `instanceOf` (b), `owl:equivalentClass` (c), `owl:disjointWith` (d), `rdfs:domain` (e) oraz `rdfs:range` (f)

3.9. Analiza obiektowa

3.9.1. Diagram klas i pakietów



Rys. 3.7. Diagram pakietów

Na rysunku Rys. 3.7 przedstawiono diagram pakietów i relacje zachodzące pomiędzy nimi. Wyróżniono sześć głównych pakietów, które zostaną opisane niżej. Klasy zostały umieszczone w odpowiednich pakietach zgodnie z zachowaniem wzorca MVC (ang. Model View Controller). Do nazw pakietów został dodany przedrostek "org.pg.eti.kask.sova".

Symbol pakietu : P001

Nazwa pakietu : options

Opis : Pakiet zawierający klasy z polami opisującymi różne (modyfikowalne) ustawienia wizualizacji takie jak: kolory, grubość linii itp.

Symbol pakietu : P002

Nazwa pakietu : nodes

Opis : Pakiet z klasami odpowiedzialnymi za wizualizację i przechowywanie danych o wierzchołkach. Każdy rodzaj wierzchołka jest odzwierciedlany przez inną klasę

Symbol pakietu : P003

Nazwa pakietu : edges

Opis : Pakiet z klasami odpowiedzialnymi za wizualizację i przechowywanie danych o krawędziach. Każdy typ krawędzie jest instancją innej klasy.

Symbol pakietu : P004

Nazwa pakietu : visualization

Opis : Zawiera klasy obsługi wizualizacji m.in. klasę zwracającą display, klasy różnych trybów wizualizacji oraz klasy filtrów. Jest najważniejszym pakietem w aplikacji.

Symbol pakietu : P005

Nazwa pakietu : graph

Opis : Pakiet zawiera klasy konwertujące obiekty OWL API na odpowiednie struktury danych, które pozwalają na wizualizację.

Symbol pakietu : P006

Nazwa pakietu : utils

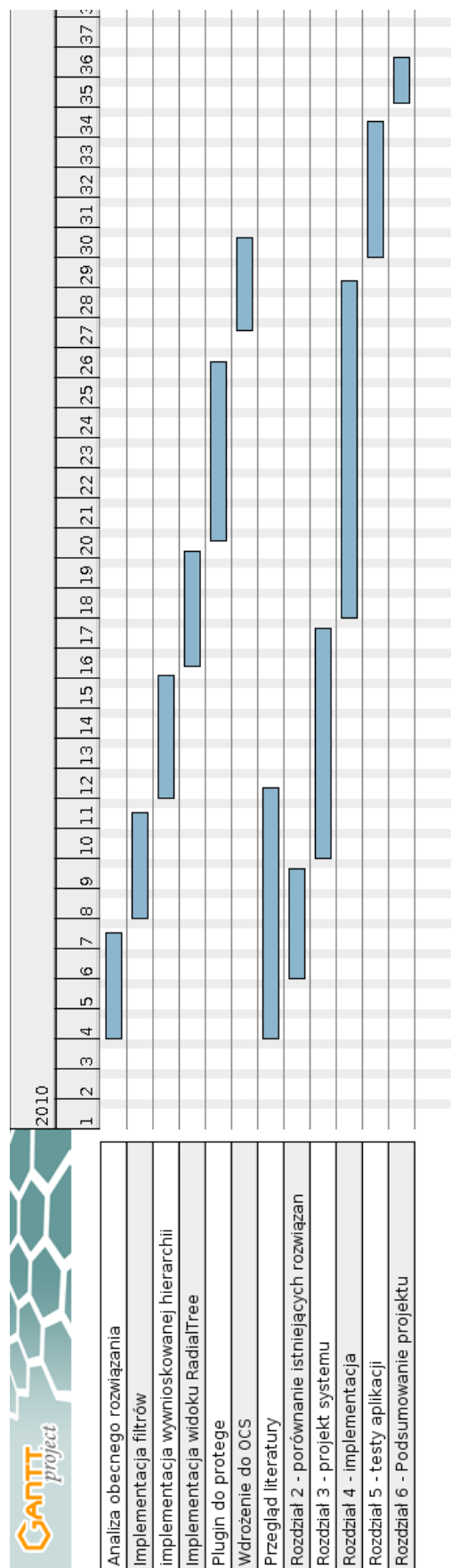
Opis : Pakiet zawiera klasy pomocnicze i dodatkowa narzędzia pozwalające m.in. na wczytanie pliku właściwości czy dostarczenie narzędzi debugiera.

3.9.2. Zalecenie tworzenia biblioteki

Nie istnieją żadne formalne zalecenia dotyczące tworzenia bibliotek JAVA. Są jednak pewne zalecenia co do stosowanych praktyk [25]:

1. **Hermetyzacja kodu.** Publiczne powinny być jedynie te klasy i metody, które są istotne dla użytkownika i z których będzie on bezpośrednio korzystał.

2. **Możliwość debugowania.** Użytkownik powinien mieć możliwość debugowania kodu biblioteki, bez konieczności znajomości każdego jej szczegółu.
3. **Przejrzystość.** Kod biblioteki powinien być odpowiednio udokumentowany za pomocą javadoc. W szczególności, bardzo dokładnie należy opisać klasy oraz metody publiczne.
4. **Łatwość użycia.** Biblioteka powinna zawierać klasy, pokazujące przykłady wykorzystania jej klas i metod.
5. **Rozszerzalność.** Struktura wewnętrzna biblioteki powinna być odpowiednio podzielona na klasy (wykorzystując klasy abstrakcyjne i interfejsy). Dzięki temu użytkownik będzie miał możliwość stworzenia własnych klas, rozszerzających funkcjonalność biblioteki.
6. **Uniwersalność.** Biblioteka powinna mieć jasno określony problem, który rozwiązuje. Wyniki powinny być podane użytkownikowi w wygodny dla niego sposób (lub na kilka sposobów), który będzie umożliwiał wykorzystanie biblioteki w różnych aplikacjach. Innymi słowy, biblioteka powinna udostępniać łatwy i przejrzysty dla użytkownika interfejs.



Rys. 3.8. Harmonogram projektu

Rozdział 4

Elementy implementacji

4.1. Wstęp

Rozdział ten jest poświęcony szczegółom implementacyjnym tworzonej biblioteki SOVA. Na początku zostaną przedstawione, poprzez opisanie i pokazanie ich architektur, dwie biblioteki użyte w projekcie OWL API oraz Prefuse.

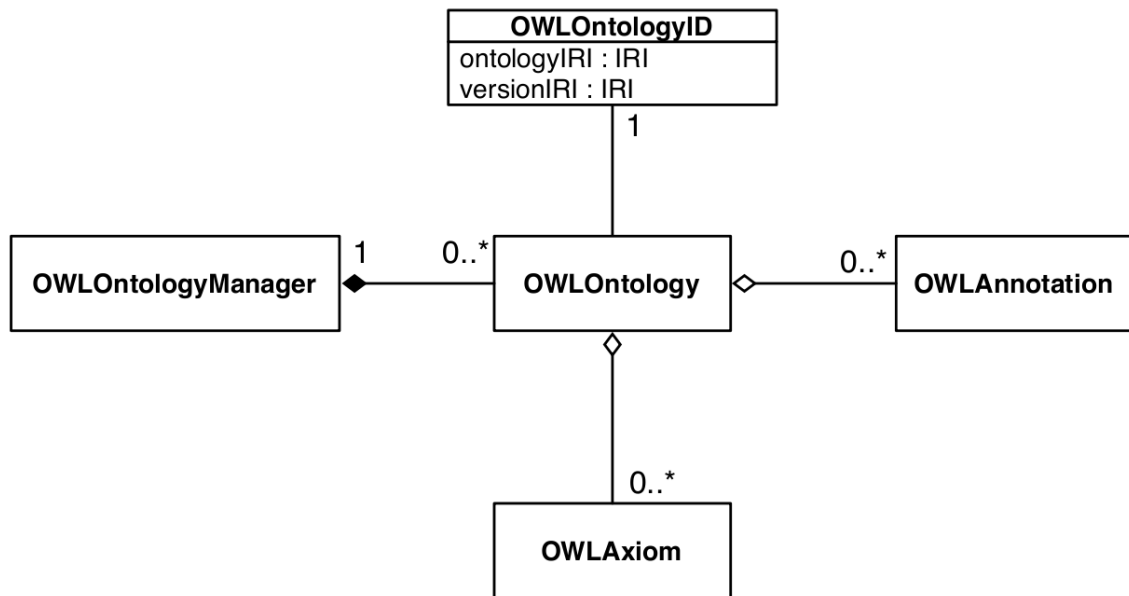
W kolejnej części pokazany zostanie sposób połączenia bibliotek z aplikacją SOVA. Zostanie zaprezentowana architektura nowej biblioteki oraz opisane zostaną algorytmy konwersji obiektów ontologii na wymagane do wizualizacji struktury.

Ostatnią część rozdziału poświęcono wdrożeniu biblioteki SOVA do konkretnych rozwiązań. Jednym z nich jest plugin do aplikacji Protégé.

4.2. Biblioteka OWL API

Podstawowym założeniem projektu jest to, że formatem wejściowym dla biblioteki wizualizującej ontologie będą obiekty OWL API, a dokładnie obiekt `OWLOntology`. OWL API jest biblioteką napisaną w javie, pozwalającą na obsługę ontologii zapisanych w języku OWL. Umożliwia tworzenie, manipulowanie oraz zapisywanie obiektów OWL. O jej użyteczności może świadczyć fakt wdrożenia jej w Protégé 4.

Najważniejszym obiektem biblioteki OWL API jest `OWLOntologyManager`. Dostarcza on metody pozwalające m.in. na tworzenie, wczytanie i zapisanie ontologii. Przechowuje również zbiór ontologii zapisywanych w obiektach `OWLOntology`. Dla wszystkich ontologii przypisanych danemu managerowi zmiany w ontologii stosowane są również do tego managera. Sposób zarządzania ontologią w OWL API został przedstawiony na rysunku 4.1



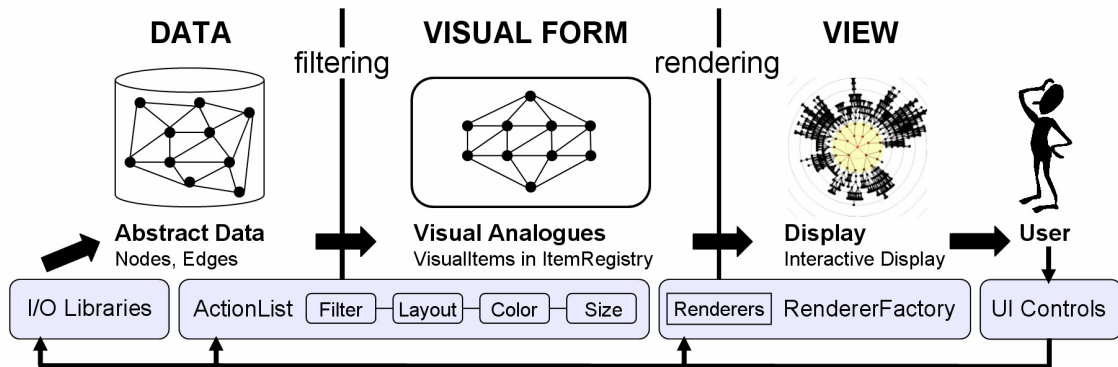
Rys. 4.1. Diagram UML pokazujący związki pomiędzy obiektami biblioteki OWL API

4.3. Biblioteka Prefuse

Głównym celem tej pracy jest wizualizacja ontologii. Musi być ona przejrzysta, a zarazem atrakcyjna dla użytkownika. Jednym z rozwiązań ułatwiających spełnienie tego wymagania jest używana w OCS biblioteka Prefuse. Prefuse jest pakietem, który dostarcza szybkich i nieskomplikowanych narzędzi do pobierania danych, manipulacji nimi oraz ich interaktywnej wizualizacji. W jego skład wchodzi:

- **prefuse.data** zawierający klasy przeznaczone do tworzenia struktur danych, zapisu i ich odczytu. Podstawową strukturą danych jest krotka. Wszystkie pozostałe struktury, np. tabele, składają się z krotek. Jednymi z bardziej rozbudowanych struktur są drzewa i grafy. Użytkownik może również definiować swoje struktury, aby w łatwy sposób edytować dane w swoim rozwiązaniu.
- **prefuse.action** celem tego modułu jest obliczenie parametrów wizualizacji. Parametrami tymi mogą być m.in. rozmieszczenie elementów grafu, jak i również nadanie kolorów. Wierzchołkowi można nadać kolor wypełnienia oraz kolor, na który ma zmienić się wypełnienie wierzchołka, w efekcie wywołania akcji, np. wskazanie elementu.
- **prefuse.visual** zawiera klasy przechowujące obiekty bezpośrednio związane z wizualizacją.
- **prefuse.controls** w skład tego pakietu wchodzi klasy związane z interaktywną wizualizacją danych, m.in. klasy i metody pozwalające na zmianę rozmiaru

elementów wizualizacji w reakcji na ruch myszą lub też klasy wywołujące odpowiednie akcje po najejchaniu bądź zaznaczeniu elementu myszą.



Rys. 4.2. Architektura logiczna biblioteki Prefuse

Proces wizualizacji danych z wykorzystaniem biblioteki Prefuse został przedstawiany na rysunku 4.2.

Kolejnymi etapami wizualizacją są:

- **Abstract Data**

Wizualizacja zaczyna się od mapowania danych na struktury dostępne w bibliotece, np. grafy. Biblioteka posiada klasy wejścia/wyjścia pozwalające, m.in. na wczytanie danych bezpośrednio z bazy danych lub pliku XML.

- **Filtering**

Filtrowanie jest procesem mapowania danych na obiekty przygotowane do wizualizacji (VisualItem). Obiekty te, poza źródłowymi danymi, przechowują informacje o położeniu elementu, jego kolorze i rozmiarze, a filtrowane VisualItem tworzą struktury danych. Często struktury te są odwzorowaniem struktur z Abstract Data, jednak ze względu na ich elastyczność mogą ulec różnego rodzaju przekształceniom. Wprowadzenie filtrów może być rozumiane jako próba implementacji wzorca MVC. W konsekwencji czego dane z tymi samymi filtrami mogą być wizualizowane na różne sposoby.

Obiekty VisualItem są tworzone i przechowywane w specjalnym rejestrze ItemRegistry, który zawiera wszystkie stany i wartości dla danej wizualizacji. Rejestr ten poprzez mechanizm buforowania danych zapewnia skalowalność rozwiązania.

- **Action**

Akcje uaktualniają obiekty VisualItem w rejestrze ItemRegistry. Mechanizm

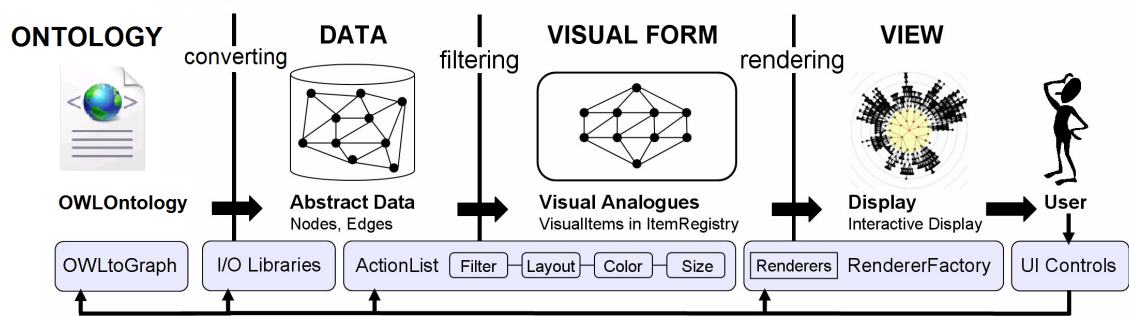
ten odpowiada za pobieranie danych, nadawanie im ustawień wizualizacji, wyrównania, przypisywaniu kolorów oraz interpolacji.

- **Rendering and Display**

Obiekty `VisualItem` są obrazowane z wykorzystaniem obiektów renderujących. Wyróżnić można rendery wizualizujące, m.in. wierzchołki, krawędzie lub tekst. Prezentacja renderowanych elementów odbywa się na specjalnym obiekcie `Display`. `Display` jest rozszerzeniem obiektu komponentu Javy z pakietu `Swing`, przyjmuje on wszystkie akcje pochodzące od użytkownika.

4.4. Integracja z OWL API i Prefuse

Biblioteka `Prefuse` [10][19], pomimo jej bogatej funkcjonalności, nie spełnia wszystkich wymagań tworzonego rozwiązania. Jednak jej architektura pozwoliła w łatwy sposób dostosować ją do wymagań tworzonej biblioteki. Proces wizualizacji danych w bibliotece `SOVA` został przedstawiony na rysunku 4.3.



Rys. 4.3. Architektura logiczna biblioteki SOVA

Poniżej przedstawiono kroki wizualizacji, opisano sposób wykorzystania elementów biblioteki `Prefuse` oraz wymagane rozszerzenia biblioteki graficznej

- **Converting** Pierwszym krokiem wizualizacji ontologii jest przekształcenie podanego obiektu OWL API na struktury danych biblioteki `Prefuse`. Każdy sposób wizualizacji (pełna wizualizacja oraz wynioskowana hierarchia) posiada inny algorytm konwersji ontologii na struktury biblioteki `Prefuse`. Algorytm (zaprezentowany na listingu nr 4.1) konwertuje obiekt `OWLontology` na struktury typu graf i jest używany przy pełnej wizualizacji. Działanie algorytmu zostało szczegółowo opisane w p. 4.5.1. W przypadku wynioskowanej hierarchii klas i bytów ontologia jest przekształcana na strukturę drzewiastą. Przekształcenie to jednak nie polega na zwykłym przeglądaniu ontologii i przepisywaniu

jej na odpowiednią strukturę danych. Wymagane jest użycie biblioteki wnioskującej Pellet. Rekurencyjny sposób budowy drzewa hierarchii został opisany w p. 4.5.2.

- **Abstract Data**

Dane przechowywane są w dwóch strukturach danych: graf i drzewo. Podstawową strukturą wykorzystywaną podczas wizualizacji jest graf. Posiada on zbiór wierzchołków i zbiór krawędzi. Struktura ta jest wykorzystywana w podstawowym sposobie wizualizacji nazywanym również pełną wizualizacją, zaś struktura drzewa przy przechowywaniu wywnioskowanej hierarchii klas i bytów.

- **Filtering**

Aplikacja korzysta z dostarczonego w bibliotece graficznej filtru na odległość. Filtr ten pozwala na obrazowanie wierzchołków oddalonych o nie więcej niż zadana liczba kroków. Filtrowanie zostało rozszerzone o filtr umożliwiający ukrywanie zadanych elementów ontologii. Filtr sprawdza jakie opcje wizualizacji zostały ustawione w statycznej klasie FilterOptions i na podstawie tych danych nadaje odpowiednie wartości obiektom VisualItem.

- **Action**

W rozwiązaniu można wyróżnić dwa rodzaje akcji: jedne związane ze zmianą położenia elementów wizualizacji, drugie odpowiedzialne za statyczne zmiany. Akcje związane z dynamiką wizualizacji mogą być zatrzymane - w ten sposób wizualizacja staje się nieruchoma.

- **Rendering and Display**

Ze względu na specyfikę obrazowanych elementów konieczne było stworzenie własnych klas renderujących. Klasy biblioteki Prefuse pozwalają tylko na rysowanie wierzchołków o kształcie prostokąta lub prostokąta z zaokrąglonymi rogami i prostych krawędzi lub krawędzi w kształcie strzałki. Właśnie dlatego powstała klasa obrazująca wierzchołki o różnych kształtach i kolorach oraz klasa pozwalająca na obrazowanie krawędzi posiadających różne groty oraz kolory. Wszystkie obiekty obrazowane są na obiekcie OVDisplay. Klasa rozszerzająca prefuse'owy Display posiada metody pozwalające użytkownikowi w łatwy sposób zobrazować zadaną ontologię. Obiektowi OVDisplay przypisane zostały metody obsługi akcji na prawy i lewy przycisk myszy.

4.5. Konwersja obiektu OWLOntology na struktury danych biblioteki Prefuse

4.5.1. Pełna wizualizacja

Kluczowym elementem wizualizacji jest pobranie z ontologii potrzebnych do wizualizacji elementów. Elementy, w szczególności klasy anonimowe występują w różnych aksjomatach. Dlatego ważne jest, aby nie dublować tych samych elementów w wizualizacji. Algorytm (listing nr 4.1) zapewnia konwersję bazowych, jak i anonimowych elementów [3].

```
1 begin
2     insert basic elements;
3     for each axiom begin
4         if property axiom begin
5             insert anonymous node;
6             insert connections between nodes;
7         end if
8         if individual axiom begin
9             insert anonymous node;
10            insert connections between nodes;
11        end if
12        if class axiom begin
13            if description axiom begin
14                start procedure InsertDescription;
15            end if
16            insert connections between nodes;
17        end if
18    end for
19    insert owl:Thing element;
20    connect not superclassed classes with owl:Thing;
21    Procedure InsertDescription
22    begin
23        if someValuesFrom or allValuesFrom or hasValue or cardinality axiom begin
24            insert property usage node;
25            insert edges;
26            start procedure InsertDescription;
27        end if
28        if setTypeAxiom begin
29            for each description node begin
30                start procedure InsertDescription;
31            end foreach
32            if class or individual begin
33                insert connection;
34            end if
35        end
36    end
```

Listing 4.1. Algorytm konwertowanie obiektu OWLOntology na obiekty kontenerowe biblioteki Prefuse

Algorytm zakłada inicjalne dodanie do wizualizacji wszystkich wierzchołków klas, właściwości oraz bytów zdefiniowanych w ontologii i następnie wstawianie łą-

czących je relacji i dodatkowych wierzchołków anonimowych oraz definiujących użycie właściwości (Some/all values from i hasValue). Po wstawieniu wierzchołków(2) przeglądana jest lista aksjomatów(3). Wśród nich wyróżnić można kilka przypadków. Dla aksjomatów definiujących właściwości (4): functional, inverseFunctional, symmetric i transitive w grafie umieszczane są odpowiednie anonimowe wierzchołki (5) i krawędzie łączące je ze stosowanym właściwościami (6). Związki pomiędzy właściwościami takie jak inverseProperty, equivalent properties i subproperty są również umieszczane zgodnie z definicją ze specyfikacji wizualizacji. Podobnie postępuje się z aksjomatami definiującymi związki między bytami (8-11) (different, allDifferent i same).

W przypadku aksjomatów uwzględniających klasy (12) algorytm musi uwzględnić fakt, że w każdym przypadku klasa może być zdefiniowana poprzez bardziej skomplikowany opis (description) (13-15). Do takich aksjomatów należą te prezentujące związki pomiędzy klasami : equivalentClasses, subclass, disjoint classes. Ponadto zależności takie mogą wystąpić w przypadku definicji poprzez klasę range i domain dla property oraz w definicjach typu classAssertion, określających klasę, do jakiej należy dany byt. Oczywiście jeśli jedna z tych zależności dotyczy samej, jawnie zdefiniowanej klasy, w grafie umieszcza się stosowna krawędź łączącą klasę z innym wierzchołkiem (16).

Opisy takie tworzone są poprzez różne klasy anonimowe zdefiniowane w wizualizacji. W przypadku opisów reprezentującym relację hasValue, someValuesFrom i allValuesFrom (23) (has Value jest relacją allValuesFrom, jeśli wierzchołkiem wynikowym jest byt, nie klasa) najpierw umieszczana jest w grafie klasa anonimowa. Jeśli aksjomat uwzględnia definicję kardynalności, to klasa ta zamieniana jest na symbol kardynalności (28) (N) wraz z odpowiednim wierzchołkiem z ograniczeniem liczbowym. Następnie do grafu wstawiana jest krawędź i stosowny wierzchołek reprezentujący użycie właściwości, krawędź łącząca to „użycie” z definicją i krawędź z grotem do wartości zdefiniowanej przez aksjomat. Jako że wartością ta może być zarówno individual jaki i klasa lub jej opis (description), należy w przypadku opisu raz jeszcze dokonać stosowanej analizy (algorytm zachowuje się tu rekurencyjnie). Dla description opisujących zbiory klas lub ich opisów: unionOf, intersectionOf, complementOf (28) należy dla każdej składowej zbioru (29) umieścić w grafie odpowiednią krawędź i, o ile klasa jest zdefiniowana opisem, przeprowadzić jego analizę, także rekurencyjnie. Jednym z typów opisu może być także klasa zdefiniowana przez aksjomat oneOf – w grafie umieszcza się stosowny wierzchołek anonimowy i powiązania z bytami.

Wszystkie te zasady stosuje się analogicznie dla typów danych (datatypes), gdzie zasady odnoszące się do klas obowiązują również dla typów danych. Większość on-

tologii nie definiuje jawnie, że klasa jest podklasą Thing, dlatego, aby zapewnić spójność grafu, dla klas nieposiadających zdefiniowanej nadklasy zakłada się, że jest nią Thing i umieszcza się stosowne krawędzie subclass (19-20). Jest to jedyne odstępstwo od zasady wizualizacji dokładnie tego, co zostało podane w ontologii, jakie algorytm musi poczynić. Dzięki temu grafy ontologii wizualizowane przy pomocy tego algorytmu można czytać niemal dokładnie tak samo jak samą ontologię zapisaną za pomocą języka OWL.

4.5.2. Wywnioskowana hierarchia klas i bytów

Wywnioskowana hierarchia klas i bytów prezentowana jest jak drzewo. Aby zobrazować ontologię w taki sposób, jest wymagany mechanizm wnioskowania (ang. reasoner). W tym celu w projekcie został użyty pellet. Pellet [23] jest otwartą biblioteką napisaną w javie, posiadającą mechanizmy wnioskowania. Obsługuje ona zarówno OWL API, jak i bibliotekę Jena. Pellet jest pierwszą biblioteką wnioskującą, która w pełni obsługuje język OWL DL oraz częściowo obsługuje OWL Full. Twórcy ontologii dążą do zapisania jej w języku OWL DL, jednak duża liczba restrykcji, jakie nakłada ten dialekt powoduje, że w rezultacie uzyskują ontologię zapisaną w OWL Full. Pellet posiada szereg heurystyk pozwalających na wykrywanie i przekształcanie ontologii z OWL Full na OWL DL. Użycie pelleta jest doskonałym sposobem na sprawdzenie spójności i poprawności ontologii. Pozwala on na znalezienie błędu niespójności ontologii poprzez znalezienie aksjomatu, który wprowadza ten błąd lub relacji dla niespójnych pojęć.

Na listingu 4.2 przedstawiono kod użyty do tworzenia drzewa wywnioskowanej hierarchii klas i bytów. Metoda buildTree() inicjuje reasoner (mechanizm wnioskujący pelleta) ontologią, którą będziemy klasyfikować(4). Korzeniem drzewa jest zawsze klasa Thing, która jest przekazywana jako parametr do rekurencyjnej metody budowania drzewa(6). Algorytm sprawdza wierzchołek przekazany jako parametr funkcji i jeśli jest on bytem, dodaje go do drzewa (55-59). Jeśli wierzchołek jest klasą, to po dodaniu go do drzewa (26-36) mechanizm wnioskujący wyszukuje wszystkie podklasy analizowanego wierzchołka (40-43) oraz wszystkie instancje tej klasy - byty (47-49). Wywołuje na nich rekurencyjne budowanie drzewa.

```
1 private void buildTree() {
2     try {
3         reasoner
4             .loadOntologies(ontologyManager.getImportsClosure(ontology));
5         OWLClass thingClass = dataFactory.getOWLThing();
6         buildTree(null, thingClass);
7         reasoner.clearOntologies();
8     } catch (UnknownOWLOntologyException e) {
9         e.printStackTrace();
10    } catch (OWLReasonerException e) {
```

```

11         e.printStackTrace();
12     }
13 }
14
15 private void buildTree(Node parentNode, OWLEntity currentEntity)
16     throws OWLReasonerException {
17     Node currentNode = null;
18     if (currentEntity instanceof OWLClass) {
19         OWLClass currentClass = (OWLClass) currentEntity;
20         if (reasoner.isSatisfiable(currentClass)) {
21             if (parentNode == null) {
22                 currentNode = tree.addRoot();
23             } else {
24                 currentNode = tree.addChild(parentNode);
25             }
26             usedClasses.add(currentClass.getURI().toString());
27             org.pg.eti.kask.sova.nodes.Node node = null;
28
29             if (currentClass.isOWLThing()){
30                 node = new org.pg.eti.kask.sova.nodes.ThingNode();
31                 node.setLabel("T");
32             } else {
33                 node = new org.pg.eti.kask.sova.nodes.ClassNode();
34                 node.setLabel(currentClass.toString());
35             }
36             currentNode.set(Constants.TREENODES, node);
37             Set<OWLClass> subClasses = OWLReasonerAdapter
38                 .flattenSetOfSets(reasoner.getSubClasses(currentClass));
39
40             for (OWLClass child : subClasses) {
41                 buildTree(currentNode, child);
42             }
43
44             Set<OWLIndividual> individuals = reasoner.getIndividuals(
45                 currentClass, true);
46
47             for (OWLIndividual ind : individuals) {
48                 buildTree(currentNode, ind);
49             }
50         }
51     }
52 } else if (currentEntity instanceof OWLIndividual) {
53     OWLIndividual currentIndividual = (OWLIndividual) currentEntity;
54
55     if (!currentIndividual.isAnonymous()) {
56         currentNode = tree.addChild(parentNode);
57         org.pg.eti.kask.sova.nodes.Node node = new IndividualNode();
58         node.setLabel(currentIndividual.toString());
59         currentNode.set(Constants.TREENODES, node);
60     }
61 }
62 }

```

Listing 4.2. Algorytm budowy wywnioskowanej hierarchii klas i bytów

Algorytm budujący drzewo wywnioskowanej hierarchii klas i bytów mechanizmu wnioskowania używa do wyszukania bytów powiązanych z danym elementem oraz wszystkich jego podklas. Podklasy te mogą niejawnie zdefiniowane.

4.6. Plik właściwości

Zaproponowane przez autora kolory wierzchołków i krawędzi wizualizacji zostały dobrane w taki sposób, aby wizualizacja była czytelna i “miła dla oka”. Niektórzy użytkownicy mogą posiadać inną percepcję wzrokową niż autor. Dlatego należy dać możliwość ustawienia przez użytkownika własnej palety kolorów. Aby zmienić kolory wizualizacji należy dostarczyć plik konfiguracyjny z własnymi ustawieniami kolorów. Plik ten składa się z kluczy przypisanych danym elementom wizualizacji oraz wartości nowo nadanego koloru. Kompletny spis dopuszczanych wartości (kluczy) został opisany w dodatku B.

Przykładowy wpis w pliku właściwości wygląda następująco:

```
node.color.thingNodeColor=#00FF00
```

W powyższym przykładzie jest ustawiany kolor dla wierzchołka thing. Kolor zapisywany jest jako RGB reprezentowany szesnastkowo.

4.7. Plugin wizualizujący ontologie do Protégé

Plugin do Protégé powstał, aby “osadzić” bibliotekę SOVA w wymagającym środowisku. Ponieważ Protégé jest bardzo popularnym edytorem ontologii, to powstały i upubliczniony plugin trafi do specjalistów zajmujących się ontologiami na co dzień i to oni będą mogli wyrazić opinie na temat tego rozwiązania. Rozwiązanie zostało upowszechnione i opisane na stronach wiki należących od autorów Protégé. Obecnie wydany na licencji LGPL plugin w wersji 0.6.0 można odnaleźć pod adresem: <http://protegewiki.stanford.edu/wiki/SOVA>.

Wykorzystanie w najnowszej wersji Protégé biblioteki OWL API znacznie ułatwiło jego integrację z powstałą biblioteką wizualizującą ontologie. Napisanie pluginu pozwoliło również na sprawdzenie, czy API powstałej biblioteki posiada wszystkie wymagane funkcje oraz czy rozwiązanie może być łatwo modyfikowalne w używanym go środowisku. Czyli zostało sprawdzone spełnienie jednego z wymagań stawianych bibliotece.

Plugin w wersji binarnej został spakowany do pliku jar. Plik ten, poza kodem biblioteki, załącza użytą bibliotekę prefuse oraz 3 pliki konfiguracyjne: plugin.xml, viewconfig.xml oraz MANIFEST.MF. Biblioteka OWL API nie jest załączona do pluginu, ponieważ posiada już ją edytor Protégé.

Protégé 4.0 umożliwia dodanie kilku typów pluginów. Najczęściej używanym jest ViewComponent - implementuje on interfejs View, który pozwala na umieszczenie

okna pluginu w dowolnej z zakładek. Sama zakładka też jest rodzajem pluginu - WorkspaceTab. Implementacja pluginu będącego zakładką nie wymaga tworzenia dodatkowych klas, jeśli plugin nie wymaga konfiguracji rozmieszczenia i wyrównania okien znajdujących się w zakładce. W przeciwnym razie jest wymagana klasa obsługi wyrównań oraz plik viewconfig.xml zawierający informacje o rozmieszczeniu okienek.

Stworzony plugin, wizualizujący ontologię z wykorzystaniem biblioteki SOVA, zawiera 2 typy pluginów: ViewComponent oraz WorkspaceTab. Rodzaje użytych pluginów i ich opis zostały zawarte w pliku konfiguracyjnym plugin.xml, który został przedstawiony na listingu nr 4.3. Plugin zakładkowy (WorkspaceTab) (17-24) wyświetla 2 rodzaje wizualizacji (zwykłą (2-8) i wywnioskowaną hierarchię(9-15)), które są pluginami typu ViewComponent i są umieszczone w nim jako 2 zakładki. Sposób rozmieszczenia elementów zakładki został skonfigurowany w pliku viewconfig.xml.

```

1 <plugin>
2     <extension id="pluginSova"
3         point="org.protege.editor.core.application.ViewComponent">
4         <label value="PG_ETI_SOVA_-_Visualization" />
5         <class value="org.pg.eti.kask.ont.pluginSova.SovaVisualization" />
6         <headerColor value="@org.protege.classcolor" />
7         <category value="@org.protege.classcategory" />
8     </extension>
9     <extension id="pluginSovaTree"
10        point="org.protege.editor.core.application.ViewComponent">
11        <label value="PG_ETI_SOVA_-_Hierarchy_Tree_Vis" />
12        <class value="org.pg.eti.kask.ont.pluginSova.HierarchyTreeVis" />
13        <headerColor value="@org.protege.classcolor" />
14        <category value="@org.protege.classcategory" />
15    </extension>
16
17    <extension id="pluginSovaTab"
18        point="org.protege.editor.core.application.WorkspaceTab">
19        <label value="PG_ETI_SOVA" />
20        <class value="org.pg.eti.kask.ont.pluginSova.pluginSovaTab" />
21        <index value="X" />
22        <editorKitId value="OWLEditorKit" />
23        <defaultViewConfigFileName value="viewconfig.xml" />
24    </extension>
25
26 </plugin>

```

Listing 4.3. Zawartość pliku konfiguracyjnego plugin.xml

Rozdział 5

Testy aplikacji

5.1. Wstęp

W ostatnim etapie pracy zostaną przeprowadzone testy aplikacji. Testy zostaną przeprowadzone dla następujących cech:

1. Funkcjonalność/Functionality – dopasowanie systemu do potrzeb funkcjonalnych, stopień pokrycia wymaganych funkcji, łatwość orientowania się w sposobie działania systemu, łatwość sprawdzenia poprawności działania systemu,
2. Wydajność/Performance – zbiór cech związanych z osiąganymi systemu (szybkość działania systemu, szybkość komunikacji z użytkownikiem, odporność systemu na zmiany środowiska),
3. Wiarygodność/Dependability - stopień zaufania do systemu, niezawodność, stopień tolerancji błędów, bezpieczeństwo, stopień kontroli dostępu do systemu, zdolność do wykrywania i identyfikacji błędów w systemie,
4. Użyteczność/Usability - wysiłek, który musi być włożony w nauczanie się programu, w jego użycie, przygotowanie danych wejściowych i interpretację danych wyjściowych.

5.2. Testy funkcjonalne

Testy funkcjonalne pozwalają na sprawdzenie zgodności oprogramowania z dokumentem specyfikacji wymagań. Sprawdzają realizację poszczególnych funkcji oprogramowania. Ten typ testów jest testem czarnej skrzynki. Osoba testująca nie ana-

lizuje kodu, czy architektury oprogramowania, interesują ją tylko dostarczona funkcjonalność testowanej aplikacji.

Poniżej przedstawiono testy funkcjonalne biblioteki SOVA. Testy zostały podzielone na dwie kategorie (zależne do użytkownika systemu): wymagania funkcjonalne związane z wizualizacją oraz wymagania funkcjonalne stawiana przez programistę używającego biblioteki. Dla każdej kategorii stworzono zestaw przypadków testowych. W tabeli 5.1 oraz 5.2 zamieszczono tematykę każdego z przypadków i wyniki lub odpowiedzi na pytania zawarte w przypadkach testowych.

5.2.1. Testy funkcjonalne API dla programisty

Przypadki testowe tej części dotyczą funkcji wymaganych przez programistów implementujących bibliotekę SOVA w swoich rozwiązaniach. Zostały one przeprowadzone podczas implementacji pluginu do Protégé oraz wdrożenia do systemu OCS.

Tab. 5.1. Przeprowadzone testy API programistycznego

Opis testu	Wynik
Pełna wizualizacja ontologii. Test polega na napisaniu kodu pozwalającego na wizualizację obiektu OWL.	Test zakończony powodzeniem. Napisa-ny krótki kod pozwolił na obrazowanie zadanego obiektu OWLOntology.
Wizualizacja wywnioskowanej hierarchii klas i bytów. Test polega na napisaniu kodu pozwalającego na wizualizację taksonomii.	Test zakończony powodzeniem. Napisa-ny krótki kod pozwolił obrazować ontologię w oczekiwany sposób.
Intuicyjne API biblioteki pozwalające na szybką wizualizację ontologii.	Dzięki zachowaniu norm nazewnictwa w języku Java, API biblioteki jest intuicyjne. Aby dokonać wizualizacji ontologii w tworzonym przez programistę środowisku, należy stworzyć tylko obiekt OVDisplay i wywołać na nim metodę generateGraphFromOWL lub generateTreeFromOWL dla wywnioskowanej hierarchii.
Zapis informacji z strumienia błędów do pliku.	Test zakończony powodzeniem. Wszystkie komunikaty debugiera pojawiły się w pliku.

Zmiana koloru wizualizowanych elementów poprzez użycie pliku properties	Test zakończony powodzeniem. Zostały wczytane wszystkie kolory zawarte w pliku properties.
---	--

5.2.2. Testy funkcjonalne związane z wizualizacją

Testy związane z wizualizacją zostały przeprowadzone przy użyciu edytora Protégé 4.0.1 oraz napisanego pluginu do wizualizacji SOVA. Przeprowadzone testy pozwoliły sprawdzić poprawności funkcji wizualizujących biblioteki SOVA zawartych w specyfikacji wymagań funkcjonalnych.

Tab. 5.2. Przeprowadzone testy funkcjonalne

Opis testu	Wynik
Zmiana trybu wizualizacji na Radial-Tree.	Test zakończony powodzeniem
Ustawienie odległości od zaznaczonego wierzchołka na 1.	Test zakończony powodzeniem. Zostały wyświetlone elementy oddalone o 1 od zaznaczonego.
Wyłączenie wizualizacji klas i elementów bezpośrednio powiązanych z klasami.	Test zakończony powodzeniem. Klasy i związki pomiędzy nimi nie pojawiły się na wizualizacji.
Wyłączenie wizualizacji bytów i elementów bezpośrednio powiązanych z nimi.	Test zakończony powodzeniem. Byty i związki pomiędzy nimi nie pojawiły się na wizualizacji.
Wyłączenie wizualizacji właściwości i elementów bezpośrednio powiązanych z właściwościami.	Test zakończony powodzeniem. Właściwości i związki pomiędzy nimi nie pojawiły się na wizualizacji.

5.3. Testy wydajnościowe

Wizualizacja ontologii jest skomplikowanym procesem. Algorytm przedstawiony w poprzednim rozdziale przegląda ontologię kilkakrotnie, wyszukując elementy zapisane w aksjomatach. Następnie elementy te są rozmieszczane w strukturach danych, filtrowane i przekazywane algorytmowi rozmieszczenia danych. Algorytm Radial-TreeLayout, rozmieszczający dane, posiada złożoność obliczeniową $O(N \log N)$ oraz $O(E)$, gdzie N to liczba wierzchołków, E to liczba krawędzi. Złożoność obliczeniowa całego procesu wizualizacji może być duża. Dlatego należy wykonać testy wydaj-

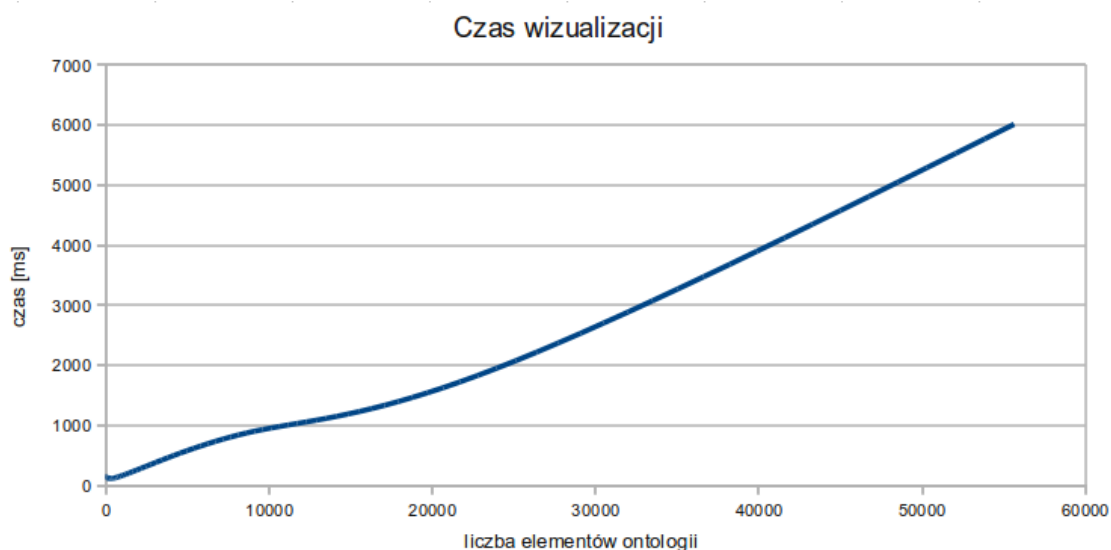
nościowe, aby upewnić się, że aplikacja jest skalowalna i potrafi wizualizować duże ontologie w przystępnym czasie.

Testy zostały przeprowadzone dla 6 ontologii posiadających różne liczby elementów. Ontologie te zostały opisane w tabeli 5.3. Tabela ta przedstawia nazwę ontologii, objętość pliku owl, z którego została wczytana. Liczbę wizualizowanych elementów ontologii, czyli liczbę wierzchołków i krawędzi użytą podczas obrazowania ontologii przy użyciu zaprojektowanych elementów graficznych. Ostatnia kolumna tabeli przedstawia czas wizualizowania ontologii, potrzebny na przetworzenie danego obiektu wejściowego OWLOntology na odpowiednie struktury danych, nadanie wartości koloru i kształtu elementom, rozmieszczenie ich algorytmem RadialTree i narysowanie grafu na obiekcie display. Czas wczytania ontologii nie jest liczony, ponieważ biblioteka jako dane wejściowe otrzymuje wczytany już obiekt owl. Obiekt ten może pochodzić nie tylko z pliku, ale np. z bazy danych.

Tab. 5.3. Pliki ontologii użyte podczas testowania

Nazwa ontologii	Rozmiar pliku [KB]	Liczba elementów	Czas [ms]
Pizza2.owl	7,24	122	170
Pizza.owl	121	1 398	203
securityontology.owl	517	4 883	353
CL.owl	1 171	9 093	900
FBsp.owl	3 159	23 917	1 947
interpro2go.owl	9 121	55 640	6 013

Na wykresie (Rys. 5.1) został przedstawiony czas wizualizacji ontologii w zależności od liczby wizualizowanych elementów. Z wykresu możemy odczytać, że czas ten zależy liniowo od liczby wizualizowanych elementów. Liczba wizualizowanych elementów nie jest równa liczbie elementów zawartych w pliku ontologii, ponieważ niektóre aksjomaty obrazowane są za pomocą kilku elementów graficznych. Np. związek sameAs łączy dwa byty, używając trzech elementów wizualizacji: dwóch linii i jednego wierzchołka. Należy jednak zauważyć, że liczba elementów potrzebna do zobrazowanie danego aksjomatu jest stała, więc liczba elementów i liczna wizualizowanych elementów zależą od siebie w sposób stały. Powyższe analiza udowadnia, że czas potrzebny na wizualizację zależy liniowo od liczby elementów zawartych w ontologii.



Rys. 5.1. Czas wizualizacji ontologii

5.4. Testy wiarygodności

Ten rodzaj testów pozwolił sprawdzić kompletność i poprawność wizualizacji. Testowanie pozwoliło wykryć czy wszystkie elementy ontologii są poprawnie wizualizowane.

Podczas testowania zostały wczytane ontologie o niewielkiej liczbie elementów. Dla każdej z nich została sprawdzona poprawność wizualizacji poprzez porównanie obrazu ontologii z jej zapisem w pliku *.owl. Poniżej opisane zostały testy oraz przedstawiono ich wyniki.

Test 1 - ontologia win

W teście została użyta mała ontologia posiadająca kilka klas i bytów. Między klasami występowały relacje m.in. subClass, disjointClass oraz oneOf. Ontologia została poprawnie zobrazowana. Żaden z elementów nie został pominięty, nie pojawiła się też żadna informacja w logu. Wynik testu jest pozytywny.

Test 2 - zmodyfikowana ontologia pizzy

Ontologia pizzy posiada większość elementów zdefiniowanych w języku OWL DL: klasy, właściwości, byty, klasy anonimowe, kardynalność oraz różnego rodzaju relacje i związki. Ontologia została poprawnie zobrazowana, wszystkie elementy pojawiły się na ekranie. W logu pojawiła się informacja o pominięciu kilku adnotacji

dotyczących aksjomatów. Adnotacje te nie mają wpływu na wygląd wizualizacji, są tylko dodatkowymi informacjami. Wynik testu został uznany za pozytywny.

Test 3 - ontologia bezpieczeństwa

Ontologia bezpieczeństwa posiada wszystkie elementy języka OWL DL. Wizualizacja tej ontologii nie zawiera, zdefiniowanych w pliku *.owl, elementów DataType. Informacja ta pojawiła się w logu. Ponieważ elementy DataType nie są bardzo istotne przy wizualizacji, a w logu pojawiła się odpowiednia informacja, wynik testu jest pozytywny.

5.5. Testy użyteczności

Testy użyteczności zostały przeprowadzone z pomocą dwóch użytkowników posiadających ogólną wiedzę o ontologiach i znających język OWL. Do testów został użyty edytor OCS z opcją wizualizacji SOVA. Każdy użytkownik miał już styczność z tą aplikacją i potrafił ją obsługiwać. Podczas testu testerzy dostali arkusz z pytaniami, które pozwalają ocenić intuicyjność interfejsu oraz jakość wizualizacji dla dwóch, różniących się rozmiarem ontologii. Wyniki testu oraz pytanie skierowane do testerów zostały zaprezentowane w tabeli 5.4.

Tab. 5.4. Wyniki testów użyteczności

Pytanie	Tester 1	Tester 2
Oceń intuicyjność interfejsu pełnej wizualizacji w skali 0-5.	4	5
Oceń intuicyjność interfejsy wywnioskowanej hierarchii w skali 0-5.	5	5
Oceń intuicyjność opcji filtrowania w skali 0-5.	4,5	4
Oceń dobór kolorów w skali 0-5.	4	4
Oceń czytelność i przejrzystość pełnej wizualizacji dla ontologii pizza2.owl w skali 0-5.	4	5
Oceń czytelność i przejrzystość pełnej wizualizacji dla ontologii pizza.owl w skali 0-5.	3	3
Oceń czytelność wizualizacji wywnioskowanej hierarchii dla ontologii pizza.owl w skali 0-5.	5	5

5.6. Podsumowanie testów

Testy pokazały, że w aplikacji poprawnie zostały zaimplementowane podstawowe funkcje. Bardzo dobrze wypadły testy wydajnościowe, podczas których udowodniono liniową złożoność obliczeniową względem liczby elementów ontologii. Przychylną opinię uzyskał interfejs aplikacji, który został pozytywnie oceniony przez użytkowników. Martwić może fakt nieobrazowania wszystkich elementów ontologii. Testy wykazały, iż element `DataType` nie jest wizualizowany, czyli nie zostało spełnione jedno z wymagań odnoszące się do kompletności wizualizacji. Na szczęście informacja o braku wizualizacji tego elementu pojawiła się w logu. Niską notę uzyskała również czytelność i przejrzystość wizualizacji dużych ontologii.

Rozdział 6

Podsumowanie

Podstawowym celem pracy było stworzenie modułu wizualizującego ontologie zapisane w języku OWL. Cel ten został osiągnięty dzięki dobrze wykonanemu projektowi architektury systemu oraz gruntownej analizie rekomendacji W3C dotyczących ontologii i języka OWL. Efektem pracy jest biblioteka SOVA, nowy moduł wizualizacji w edytorze OCS oraz plugin do aplikacji Protégé.

Wdrożenie rozwiązania w systemie OCS znacznie zwiększyło jego atrakcyjność. W tabeli 6.1 jeszcze raz przedstawiono wyniki przeprowadzonego porównania aplikacji pozwalających na wizualizację ontologii. Tabela została rozszerzona o testy dla edytora OCS z nowym modułem wizualizacji SOVA. Możemy zauważyć, że ocena wizualizacji dla OCS wzrosła z 0.34 do 0.83. Co klasyfikuje go na pierwszym miejscu.

Bardzo dobrym pomysłem okazało się stworzenie pluginu do edytora Protégé. Wydany na licencji LGPL plugin, został umieszczony i opisany na stronie <http://protegewiki.stanford.edu>. Ciągłe pobierane ze strony projektu rozwiązanie dobrze promuje uczelnię oraz katedralny system OCS.

Prace nad stworzeniem rozwiązania do wizualizacji ontologii trwały dziewięć miesięcy. W tym czasie powstało ok. 8 tysięcy linii kodu. Bardzo trudnym elementem okazała się realizacja zadań zgodnie z zaplanowanym harmonogramem. Nie wszystkie elementy pracy zostały zrealizowane w wyznaczonym terminie, a całość prac wydłużyła się o miesiąc. Przyczyną opóźnień m.in. jest słaba dokumentacja bibliotek OWL API i Prefuse.

Ograniczenia czasowe spowodowały, iż nie udało się zrealizować wszystkich wymagań i założeń pracy. Nie zostały wykonane następujące elementy:

1. Brak wizualizacji DataType, co zostało wykryte podczas testowania aplikacji.
2. Brak opcji wyszukiwania elementów ontologii na grafie.

Poza elementami, które nie zostały zrealizowane podczas pracy, autor zauważa możliwości dalszego rozwoju oraz rozszerzenia aplikacji o następujące funkcje:

1. Zmiana biblioteki OWLAPI na najnowszą wersję 3.0. Zmiana powinna być dokonana zarówno w bibliotece SOVA, jak i w kodzie OCS.
2. W module wizualizacji OCS oraz pluginie dodanie możliwość edycji ontologii z poziomu wizualizacji. Biblioteka SOVA daje możliwość podpięcia dodatkowej obsługi zdarzeń, dzięki temu możemy np. na zdarzenia kliknięcia prawym przyciskiem myszy na element ontologii, wywołać dodatkowe okno lub menu.
3. Zwiększenie przejrzystości wizualizacji poprzez dynamiczny dobór optymalnych ustawień wizualizacji (np. długość krawędzi) w zależności od liczby obrazowanych elementów wczytanej ontologii.

Tab. 6.1. Porównanie rozwiązań do wizualizacji ontologii




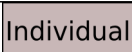

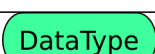

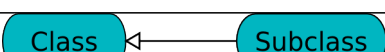
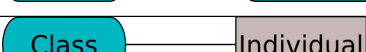
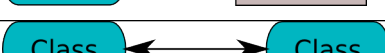

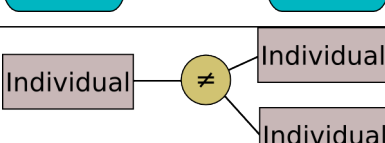

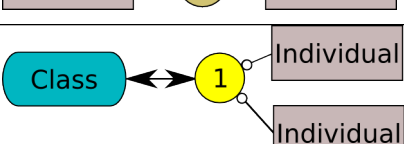
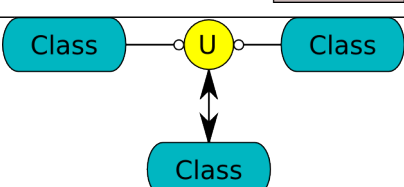
Kryterium	OntoVis	Jambalaya	Growl	OCS	OCS + SOVA
Kompletność wizualizacji (max 30 punktów)					
Sposób wizualizacji klas	2	2	2	2	2
Sposób wizualizacji bytów	2	2	2	0	2
Wizualizacja klas ananimowych	0	0	2	0	2
Wizualizacja relacji	4	3	10	2	11
Wizualizacja właściwości	2	2	2	0	2
Wizualizacja związków odnoszących się do właściwościami	0	0	4	0	7
Wizualizacja pozostałych elementów ontologii	0	3	1	0	4
Łącznie dla kategorii	0,33	0,4	0,77	0,13	1,00
Przejrzystość wizualizacji (max 8 punktów)					
Czytelność ontologii	3	4	1	2	3
Dostępność filtrów	2	3	0	0	3
Łącznie dla kategorii	0,625	0,875	0,125	0,25	0,75
Sposoby wizualizacji (max 4 punkty)					
Liczba sposobów wizualizacji	0	3	1	1	2
Wizualizacja wywnioskowanej hierarchii	0	0	0	1	1
Łącznie dla kategorii	0	0,75	0,25	0,5	0,75
Użyteczność (max 11 punktów)					
Użyteczność	4	5	5	5	5
Dostępność pomocy	2	2	2	0	1
Licencja	2	1	2	2	2
Łącznie dla kategorii	0,73	0,73	0,82	0,64	0,73
Ocena rozwiązania	0,42	0,66	0,5	0,34	0,83

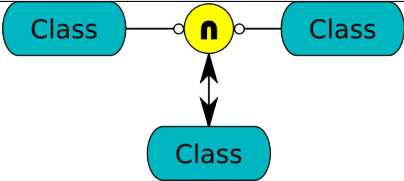
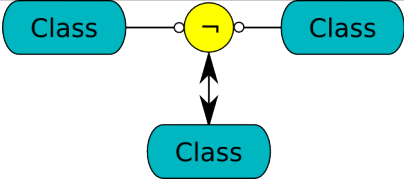
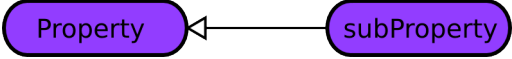
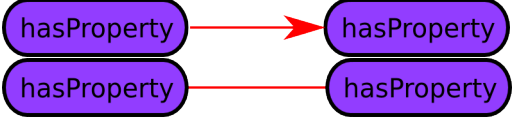





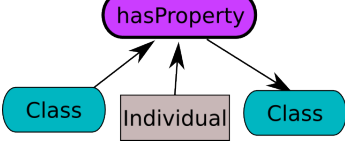
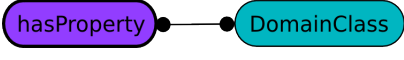
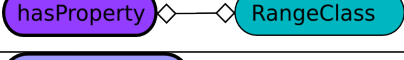


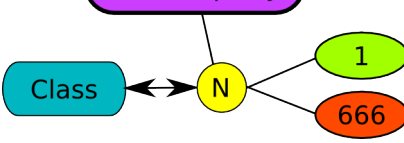
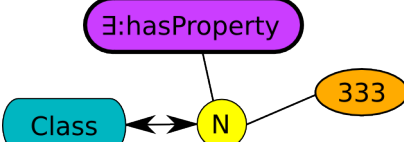
Bibliografia

- [1] Bechhofer S., Harmelen F., Hendler J., Horrocks I., McGuinness D. L., Patel-Schneider P. F., Stein L. A., *OWL Web Ontology Language Reference W3C Recommendation*, luty 2004, <http://www.w3.org/TR/owl-ref/>.
- [2] Boiński T., *Architektura portalu dziedzinyowego*, KASKBOOK 2007, Politechnika Gdańska, Gdańsk, Białogóra, 2007.
- [3] Boiński T., Jaworska A., Kleczkowski R., Kunowski P., *Ontology Visualization*, Politechnika Gdańska, 2nd International Conference on Information Technology, Gdańsk, czerwiec 2010
- [4] Ernst N., Lintern R., Perrin D., Storey M., *Visualization and Protégé*, University of Victoria, Kanada, lipiec 2005.
- [5] Gasević D., Djurić D., Devedzić V., *Model Driven Engineering and Ontology Development*, Springer, 2009.
- [6] Goczyła K., *Wnioskowanie w ontologiach opartych na logice opisowej*, KASKBOOK 2007, Politechnika Gdańska, Gdańsk, Białogóra, 2007.
- [7] Goczyła K., Zawadzka T., *Ontologie w Sieci Semantycznej*, Wydział Elektroniki, Telekomunikacji i Informatyki, Politechnika Gdańska, 2006
- [8] Grimm S., Hitzler P., Abecker A., *Knowledge Representation and Ontologies*, Niemcy, 2007.
- [9] Gruber T. R., *Toward Principles for the Design of Ontologies Used for Knowledge Sharing*, Stanford Knowledge Systems Laboratory, Stanford University, 1993.
- [10] Heer J., Card S. K., Landay J. A., *prefuse: a toolkit for interactive information visualization*. CHI 2005, 2005.
- [11] Horridge M., Bechhofer S., *The OWL API: A Java API for Working with OWL 2 Ontologies*, University of Manchester, sierpień 2009.

- [12] Horridge M., Bechhofer S., Noppens O., *Igniting the OWL 1.1 Touch Paper: The OWL API*, University of Manchester, czerwiec 2007.
- [13] Jankowski A., *Praca dyplomowa magisterska - Platforma do edycji ontologii z dostępem przez WWW*, Politechnika Gdańska, Gdańsk, grudzień 2008.
- [14] Jędruch A., *Rozwój języków ontologii*, KASKBOOK 2007, Politechnika Gdańska, Gdańsk, Białogóra, 2007.
- [15] Katifori A., Halatsis C., Lepouras G., Vassilakis C., Giannopoulou E. G., *Ontology visualization methods - a survey*, ACM Comput. Surv, 2007.
- [16] Krivov1 S., Villa F., Williams R., Wu X., *On Visualization of OWL Ontologies*, lipiec 2006.
- [17] Lintern R., Storey M., *Jambalaya express: on demand knowledge visualization*, University of Victoria, Kanada, lipiec 2005.
- [18] Manola F., Miller E., *RDF Primer W3C Recommendation*, 2004, <http://www.w3.org/TR/rdf-primer/>.
- [19] Nazimek P., *Prefuse*, Software Developer's Journal 3/2008.
- [20] *OntoSphere3D*, 2004, <http://ontosphere3d.sourceforge.net/>.
- [21] *OWL 2 Web Ontology Language Profiles W3C Recommendation*, 27 października 2009, <http://www.w3.org/TR/owl2-profiles/>.
- [22] Sintek M., *OntoViz*, 2007, <http://protegewiki.stanford.edu/wiki/OntoViz>.
- [23] Sirin E., Parsia D., Grau B. C., Kalyanpur A., Katz Y., *Pellet: A Practical OWL-DL Reasoner*, University of Maryland, 2005.
- [24] Smith M. K., Welty C., McGuinness D. L., *OWL Web Ontology Language Guide W3C Recommendation*, luty 2004, <http://www.w3.org/TR/owl-guide/>.
- [25] Travis G., *Build your own Java library*, maj 2001, [http://www.digilife.be/quickreferences/PT/Build your own Java library.pdf](http://www.digilife.be/quickreferences/PT/Build%20your%20own%20Java%20library.pdf).

Dodatek A - Projekt wizualizacji

Identyfikator:	Nazwa	Wizualizacja
PW001:	Thing	
PW002:	Nothing	
PW003:	Class	
PW004:	Individual	
PW005:	Property	
PW006:	Datatype	
PW007:	Anonymous Class	
PW008:	Subclass	
PW009:	instanceOf	
PW010:	equivalentClass	
PW011:	disjointWith	
PW012:	differentFrom / allDifferent	
PW013:	sameAs	
PW014:	oneOf	
PW015:	unionOf	

PW016:	intersectionOf	
PW017:	complementOf	
PW018:	subProperty	
PW019:	inverseOf (property)	
PW020:	equivalentProperty	
PW021:	functionalProperty	
PW022:	inverseFunctionalProperty	
PW023:	symmetricProperty	
PW024:	transitiveProperty	
PW025:	hasProperty	
PW026:	domain	
PW027:	range	
PW028:	allValuesFrom	
PW029:	someValuesFrom	
PW030:	minCardinality / maxCardinality	
PW031:	cardinality	

Dodatek B - Plik konfiguracyjny

Klucz :	Opis
node.color.thingNodeColor	Kolor węzła reprezentującego klasę "Thing"
node.color.classNodeColor	Kolor węzłów reprezentujących definicje klas
node.color.nothingNodeColor	Kolor węzła reprezentującego klasę "Nothing"
node.color.individualNodeColor	Kolor węzłów reprezentujących instancje klas (OWL Individual)
node.color.differentNodeColor	Kolor węzłów oznaczających relację DifferentFrom lub AllDifferent między wystąpieniami klas (OWL Individual)
node.color.sameAsNodeColor	Kolor węzłów oznaczających relację OWL SameAs między wystąpieniami klas (OWL Individual)
node.color.propertyNodeColor	Kolor węzłów reprezentujących definicje predykatów (OWL Property)
node.color.someValuesFromNodeColor	Kolor węzłów Property typu SomeValuesFrom
node.color.allValuesFromNodeColor	Kolor węzłów Property typu AllValuesFrom
node.color.dataTypeNodeColor	Kolor węzłów OWL DataType
node.color.anonymousClassNodeColor	Kolor węzłów reprezentujących różnorakie klasy anonimowe
node.color.cardinalityValueNodeColor	Kolor węzłów reprezentujących dokładne ograniczenie kardynalności
node.color.minCardinalityValueNodeColor	Kolor węzłów reprezentujących minimalne ograniczenie kardynalności

node.color.maxCardinalityValueNodeColor	Kolor węzłów reprezentujących maksymalne ograniczenie kardynalności
node.color.informationNodeColor	Kolor węzłów oznaczających właściwości predykatów (OWL Property)
edge.color.subClassEdgeColor	Kolor krawędzi subClassEdge
edge.color.subPropertyEdgeColor	Kolor krawędzi subPropertyEdge
edge.color.edgeColor	Kolor zwykłych krawędzi (bez grotów)
edge.color.propertyEdgeColor	Kolor krawędzi oznaczających relacje między Property a klasą
edge.color.domainEdgeColor	Kolor krawędzi łączących definicję property z jego domeną
edge.color.rangeEdgeColor	Kolor krawędzi łączących definicję property z jego przestrzenią (OWL Range)
edge.color.disjointEdgeColor	Kolor krawędzi łączących klasy rozłączne
edge.color.equivalentEdgeColor	Kolor krawędzi łączących klasy równoważne (OWL Equivalent)
edge.color.equivalentPropertyEdgeColor	Kolor krawędzi łączących predykaty (OWL Property) równoważne (OWL Equivalent)
edge.color.functionalEdgeColor	Kolor krawędzi łączących definicję Property z jego właściwościami np. functional, symmetric
edge.color.instancePropertyEdgeColor	Krawędz łącząca wystąpienie Property z jego definicją
edge.color.inverseOfEdgeColor	Kolor krawędzi łączących predykat (OWL Property) odwrotny (OWL InverseOf) do zadanego
edge.color.instanceOfEdgeColor	Kolor krawędzi instancji klasy (OWL InstanceOf)
edge.color.inverseOfMutualEdgeColor	Kolor krawędzi łączących predykaty (OWL Property) wzajemnie odwrotne (OWL InverseOf)
edge.color.operationEdgeColor	Kolor krawędzi oznaczających operacje, w wyniku których powstają klasy anonimowe np. unia, przecięcie

Dodatek C - Zawartość płyty CD

Do niniejszej pracy załączono płytę CD zawierającą:

- kopię pracy w wersji elektronicznej,
- kod źródłowy biblioteki SOVA wraz z użytymi bibliotekami,
- opis API biblioteki w postaci javadoców,
- kod źródłowy pluginu do aplikacji Protégé,
- plugin do aplikacji Protégé w wersji 0.6.0,
- analiza obiektowa i diagramy klas,
- porównanie rozwiązań do wizualizacji ontologii.